

PVS-Studio Documentation (single file)

1. Introduction
 - a. [Overview.](#)
 - b. [Getting acquainted with PVS-Studio code analyzer.](#)
 - c. [System Requirements.](#)
 - d. [Registration.](#)
 - e. [Release History.](#)
 - f. [Old PVS-Studio Release History \(before 4.00\).](#)
 - g. [Limitation.](#)
2. Using the program
 - a. [Common information on working with the PVS-Studio analyzer.](#)
 - b. [Working Mode.](#)
 - c. [Suppression of false alarms.](#)
 - d. [The principle of constantly using PVS-Studio in the process of project development.](#)
 - e. [Handling the diagnostic messages list.](#)
 - f. [OmniSample, a collection of examples of 64-bit and parallel errors.](#)
 - g. [Checking code with PVS-Studio from the command line \(with a Visual Studio solution file present\).](#)
 - h. [Using PVS-Studio from the command line \(without Visual Studio solution file\).](#)
 - i. [Using PVS-Studio with continuous integration systems.](#)
 - j. [Predefined PVS_STUDIO macro.](#)
 - k. [PVS-Studio FAQ.](#)
 - l. [Tips on speeding up PVS-Studio.](#)
 - m. [PVS-Studio menu commands.](#)
3. PVS-Studio Options
 - a. [Settings: General.](#)
 - b. [Settings: Common Analyzer.](#)
 - c. [Settings: Customers Specific Settings.](#)
 - d. [Settings: Detectable Errors.](#)
 - e. [Settings: Don't Check Files.](#)
 - f. [Settings: Message Suppression.](#)
 - g. [Settings: Registration.](#)
 - h. [Settings: Viva64.](#)
4. Error description
 - a. Common
 - i. [V001.](#) A code fragment from 'file' cannot be analyzed.
 - ii. [V002.](#) Some diagnostic messages may contain incorrect line number.
 - iii. [V003.](#) Unrecognized error found...
 - iv. [V004.](#) For a more precise verification, please select x64 or Itanium configuration instead of Win32.
 - v. [V005.](#) Cannot determine active configuration for project. Please check projects and solution configurations.
 - vi. [V006.](#) File cannot be processed. Analysis aborted by timeout.
 - vii. [V007.](#) Verification of CLR projects is not implemented. Incorrect diagnostics are possible.
 - viii. [V008.](#) Unable to start the analysis on this file.
 - ix. [V009.](#) Intel C++ project toolset is not supported. Select Visual C++ toolset to verify this project.
 - b. Diagnosis of 64-bit errors (Viva64)
 - i. [V101.](#) Implicit assignment type conversion to memsize type.
 - ii. [V102.](#) Usage of non memsize type for pointer arithmetic.
 - iii. [V103.](#) Implicit type conversion from memsize to 32-bit type.
 - iv. [V104.](#) Implicit type conversion to memsize type in an arithmetic expression.
 - v. [V105.](#) N operand of '?' operation: implicit type conversion to memsize type.
 - vi. [V106.](#) Implicit type conversion N argument of function 'foo' to memsize type.
 - vii. [V107.](#) Implicit type conversion N argument of function 'foo' to 32-bit type.
 - viii. [V108.](#) Incorrect index type: 'foo[not a memsize-type]'. Use memsize type instead.
 - ix. [V109.](#) Implicit type conversion of return value to memsize type.
 - x. [V110.](#) Implicit type conversion of return value from memsize type to 32-bit type.
 - xi. [V111.](#) Call of function 'foo' with variable number of arguments. N argument has memsize type.
 - xii. [V112.](#) Dangerous magic number N used.
 - xiii. [V113.](#) Implicit type conversion from memsize to double type or vice versa.
 - xiv. [V114.](#) Dangerous explicit type pointer conversion.
 - xv. [V115.](#) Memsize type is used for throw.
 - xvi. [V116.](#) Memsize type is used for catch.
 - xvii. [V117.](#) Memsize type is used in the union.
 - xviii. [V118.](#) malloc() function accepts a dangerous expression in the capacity of an argument.
 - xix. [V119.](#) More than one sizeof() operator is used in one expression.
 - xx. [V120.](#) Member operator[] of object 'foo' declared with 32-bit type argument, but called with memsize type argument.
 - xxi. [V121.](#) Implicit conversion of the type of 'new' operator's argument to size_t type.
 - xxii. [V122.](#) Memsize type is used in the struct/class.
 - xxiii. [V123.](#) Allocation of memory by the pattern "(X*)malloc(sizeof(Y))"
 - xxiv. [V124.](#) Function 'Foo' writes/reads 'N' bytes. The alignment rules and type sizes have been changed. Consider reviewing this value.
 - xxv. [V125.](#) It is not advised to declare type 'T' as 32-bit type.
 - xxvi. [V126.](#) Be advised that the size of the type 'long' varies between LLP64/LP64 data models.
 - xxvii. [V127.](#) An overflow of the 32-bit variable is possible inside a long cycle which utilizes a memsize-type loop counter.
 - xxviii. [V201.](#) Explicit conversion from 32-bit integer type to memsize type.
 - xxix. [V202.](#) Explicit conversion from memsize type to 32-bit integer type.
 - xxx. [V203.](#) Explicit type conversion from memsize to double type or vice versa.
 - xxxi. [V204.](#) Explicit conversion from 32-bit integer type to pointer type.
 - xxxii. [V205.](#) Explicit conversion of pointer type to 32-bit integer type.
 - xxxiii. [V220.](#) Suspicious sequence of types castings: memsize -> 32-bit integer -> memsize.
 - xxxiv. [V301.](#) Unexpected function overloading behavior. See N argument of function 'foo' in derived class 'derived' and base class 'base'.
 - xxxv. [V302.](#) Member operator[] of 'foo' class has a 32-bit type argument. Use memsize-type here.
 - xxxvi. [V303.](#) The function is deprecated in the Win64 system. It is safer to use the 'foo' function.
 - c. General Analysis
 - i. [V501.](#) There are identical sub-expressions to the left and to the right of the 'foo' operator.
 - ii. [V502.](#) Perhaps the '?' operator works in a different way than it was expected. The '?' operator has a lower priority than the 'foo' operator.
 - iii. [V503.](#) This is a nonsensical comparison: pointer < 0.
 - iv. [V504.](#) It is highly probable that the semicolon ';' is missing after 'return' keyword.
 - v. [V505.](#) The 'alloca' function is used inside the loop. This can quickly overflow stack.
 - vi. [V506.](#) Pointer to local variable 'X' is stored outside the scope of this variable. Such a pointer will become invalid.
 - vii. [V507.](#) Pointer to local array 'X' is stored outside the scope of this array. Such a pointer will become invalid.
 - viii. [V508.](#) The use of 'new type(n)' pattern was detected. Probably meant: 'new type[n]'.
 - ix. [V509.](#) The 'throw' operator inside the destructor should be placed within the try..catch block. Raising exception inside the destructor is illegal.
 - x. [V510.](#) The 'Foo' function is not expected to receive class-type variable as 'N' actual argument.

- xi. [V511](#). The `sizeof()` operator returns size of the pointer, and not of the array, in given expression.
- xii. [V512](#). A call of the 'Foo' function will lead to a buffer overflow or underflow.
- xiii. [V513](#). Use `_beginthreadex/_endthreadex` functions instead of `CreateThread/ExitThread` functions.
- xiv. [V514](#). Dividing size of a pointer by another value. There is a probability of logical error presence.
- xv. [V515](#). The 'delete' operator is applied to non-pointer.
- xvi. [V516](#). Consider inspecting an odd expression. Non-null function pointer is compared to null.
- xvii. [V517](#). The use of 'if (A) {...} else if (A) {...}' pattern was detected. There is a probability of logical error presence.
- xviii. [V518](#). The 'malloc' function allocates strange amount of memory calculated by 'strlen(expr)'. Perhaps the correct variant is `strlen(expr) + 1`.
- xix. [V519](#). The 'X' variable is assigned values twice successively. Perhaps this is a mistake.
- xx. [V520](#). The comma operator ',' in array index expression.
- xxi. [V521](#). Such expressions using the ',' operator are dangerous. Make sure the expression is correct.
- xxii. [V522](#). Dereferencing of the null pointer might take place. Check the logical condition.
- xxiii. [V523](#). The 'then' statement is equivalent to the 'else' statement.
- xxiv. [V524](#). It is odd that the body of 'Foo_1' function is fully equivalent to the body of 'Foo_2' function.
- xxv. [V525](#). The code containing the collection of similar blocks. Check items X, Y, Z, ... in lines N1, N2, N3, ...
- xxvi. [V526](#). The 'strcmp' function returns 0 if corresponding strings are equal. Consider examining the condition for mistakes.
- xxvii. [V527](#). It is odd that the 'zero' value is assigned to pointer. Probably meant: `*ptr = zero`.
- xxviii. [V528](#). It is odd that pointer is compared with the 'zero' value. Probably meant: `*ptr != zero`.
- xxix. [V529](#). Odd semicolon ';' after 'if/for/while' operator.
- xxx. [V530](#). The return value of function 'Foo' is required to be utilized.
- xxxi. [V531](#). It is odd that a `sizeof()` operator is multiplied by `sizeof()`.
- xxxii. [V532](#). Consider inspecting the statement of `*pointer++` pattern. Probably meant: `(*pointer)++`.
- xxxiii. [V533](#). It is likely that a wrong variable is being incremented inside the 'for' operator. Consider reviewing 'X'.
- xxxiv. [V534](#). It is likely that a wrong variable is being compared inside the 'for' operator. Consider reviewing 'X'.
- xxxv. [V535](#). The variable 'X' is being used for this loop and for the outer loop.
- xxxvi. [V536](#). Be advised that the utilized constant value is represented by an octal form.
- xxxvii. [V537](#). Consider reviewing the correctness of 'X' item's usage.
- xxxviii. [V538](#). The line contains control character 0x0B (vertical tabulation).
- xxxix. [V539](#). Consider inspecting iterators which are being passed as arguments to function 'Foo'.
 - xl. [V540](#). Member 'x' should point to string terminated by two 0 characters.
 - xli. [V541](#). It is dangerous to print the string into itself.
 - xlii. [V542](#). Consider inspecting an odd type cast: 'Type1' to 'Type2'.
 - xliii. [V543](#). It is odd that value 'X' is assigned to the variable 'Y' of HRESULT type.
 - xliv. [V544](#). It is odd that the value 'X' of HRESULT type is compared with 'Y'.
 - xlv. [V545](#). Such conditional expression of 'if' operator is incorrect for the HRESULT type value 'Foo'. The SUCCEEDED or FAILED macro should be used instead.
 - xlvi. [V546](#). Member of a class is initialized by itself: 'Foo(Foo)'.
 - xlvii. [V547](#). Expression is always true/false.
- xl. [V548](#). Consider reviewing type casting. `TYPE X[]` is not equivalent to `TYPE **X`.
- xli. [V549](#). The first argument of 'Foo' function matches it's the second argument.
 - i. [V550](#). An odd precise comparison. It's probably better to use a comparison with defined precision: `fabs(A - B) < Epsilon` or `fabs(A - B) > Epsilon`.
 - ii. [V551](#). The code under this 'case' label is unreachable.
 - iii. [V552](#). A bool type variable is being incremented. Perhaps another variable should be incremented instead.
 - liii. [V553](#). The length of function's body or class's declaration is more than 2000 lines long. You should consider refactoring the code.
 - liv. [V554](#). Incorrect use of smart pointer.
 - lv. [V555](#). The expression of the 'A - B > 0' kind will work as 'A != B'.
 - lv. [V556](#). The values of different enum types are compared.
 - lvii. [V557](#). Array overrun is possible.
 - lviii. [V558](#). Function returns the pointer to temporary local object.
 - lix. [V559](#). Suspicious assignment inside the condition expression of 'if/while' operator.
 - lx. [V560](#). A part of conditional expression is always true/false.
 - lxi. [V561](#). It's probably better to assign value to 'foo' variable than to declare it anew.
 - lxii. [V562](#). It's odd to compare a bool type value with a value of N.
 - lxiii. [V563](#). It is possible that this 'else' branch must apply to the previous 'if' statement.
 - lxiv. [V564](#). The '&' or '!' operator is applied to bool type value. You've probably forgotten to include parentheses or intended to use the '&&' or '||' operator.
 - lxv. [V565](#). An empty exception handler. Silent suppression of exceptions can hide the presence of bugs in source code during testing.
 - lxvi. [V566](#). The integer constant is converted to pointer. Possibly an error or a bad coding style.
 - lxvii. [V567](#). Undefined behavior. The variable is modified while being used twice between sequence points.
 - lxviii. [V568](#). It's odd that the argument of `sizeof()` operator is the expression.
 - lxix. [V569](#). Truncation of constant value.
 - lxx. [V570](#). The variable is assigned to itself.
 - lxxi. [V571](#). Recurring check. This condition was already verified in previous line.
 - lxxii. [V572](#). It is odd that the object which was created using 'new' operator is immediately casted to another type.
 - lxxiii. [V573](#). Uninitialized variable 'Foo' was used. The variable was used to initialize itself.
 - lxxiv. [V574](#). The pointer is used simultaneously as an array and as a pointer to single object.
 - lxxv. [V575](#). Function receives an odd argument.
 - lxxvi. [V576](#). Incorrect format. Consider checking the N actual argument of the 'Foo' function.
 - lxxvii. [V577](#). Label is present inside a switch(). It is possible that these are misprints and 'default' operator should be used instead.
 - lxxviii. [V578](#). An odd bitwise operation detected. Consider verifying it.
 - lxxix. [V579](#). The 'Foo' function receives the pointer and its size as arguments. It is possibly a mistake. Inspect the N argument.
 - lxxx. [V580](#). An odd explicit type casting. Consider verifying it.
 - lxxxi. [V581](#). The conditional expressions of the 'if' operators situated alongside each other are identical.
 - lxxxii. [V582](#). Consider reviewing the source code which operates the container.
 - lxxxiii. [V583](#). The '?' operator, regardless of its conditional expression, always returns one and the same value.
 - lxxxiv. [V584](#). The same value is present on both sides of the operator. The expression is incorrect or it can be simplified.
 - lxxxv. [V585](#). An attempt to release the memory in which the 'Foo' local variable is stored.
 - lxxxvi. [V586](#). The 'Foo' function is called twice for deallocation of the same resource.
 - lxxxvii. [V587](#). An odd sequence of assignments of this kind: `A = B; B = A;`
 - lxxxviii. [V588](#). The expression of the 'A += B' kind is utilized. Consider reviewing it, as it is possible that 'A += B' was meant.
 - lxxxix. [V589](#). The expression of the 'A -= B' kind is utilized. Consider reviewing it, as it is possible that 'A -= B' was meant.
 - xc. [V590](#). Consider inspecting this expression. The expression is excessive or contains a misprint.
 - xci. [V591](#). Non-void function should return a value.
 - xcii. [V592](#). The expression was enclosed by parentheses twice: `((expression))`. One pair of parentheses is unnecessary or misprint is present.
 - xciii. [V593](#). Consider reviewing the expression of the 'A = B == C' kind. The expression is calculated as following: `A = (B == C)`.
 - xciv. [V594](#). The pointer steps out of array's bounds.
 - xcv. [V595](#). The pointer was utilized before it was verified against `nullptr`. Check lines: N1, N2.
 - xcvi. [V596](#). The object was created but it is not being used. The 'throw' keyword could be missing.
 - xcvii. [V597](#). The compiler could delete the 'memset' function call, which is used to flush 'Foo' buffer. The `RtlSecureZeroMemory()` function should be used to erase the private data.
 - xcviii. [V598](#). The 'memset/memcpy' function is used to nullify/copy the fields of 'Foo' class. Virtual method table will be damaged by this.
 - xcix. [V599](#). The virtual destructor is not present, although the 'Foo' class contains virtual functions.
 - c. [V600](#). Consider inspecting the condition. The 'Foo' pointer is always not equal to NULL.

- ci. [V601](#). An odd implicit type casting.
- cii. [V602](#). Consider inspecting this expression. '<' possibly should be replaced with '<<'.
- ciiii. [V603](#). The object was created but it is not being used. If you wish to call constructor, 'this->Foo::Foo(...)' should be used.
- civ. [V801](#). Decreased performance. It is better to redefine the N function argument as a reference. Consider replacing 'const T' with 'const .. &T' / 'const .. *T'.
- cv. [V802](#). On 32-bit/64-bit platform, structure size can be reduced from N to K bytes by rearranging the fields according to their sizes in decreasing order.
- cvi. [V803](#). Decreased performance. It is more effective to use the prefix form of ++it. Replace iterator++ with ++iterator.
- cvi. [V804](#). Decreased performance. The 'Foo' function is called twice in the specified expression to calculate length of the same string.
- cvi. [V805](#). Decreased performance. It is inefficient to identify an empty string by using 'strlen(str) > 0' construct. A more efficient way is to check: str[0] != '\0'.
- cix. [V806](#). Decreased performance. The expression of strlen(MyStr.c_str()) kind can be rewritten as MyStr.length().
- cx. [V807](#). Decreased performance. Consider creating a pointer/reference to avoid using the same expression repeatedly.
- d. Diagnosis of parallel errors (VivaMP)
 - i. [V1000](#). Did you forget to enable the /openmp compiler option?
 - ii. [V1001](#). Missing 'parallel' keyword.
 - iii. [V1002](#). Missing 'omp' keyword.
 - iv. [V1003](#). Missing 'for' keyword. Each thread will execute the entire loop.
 - v. [V1004](#). Nested parallelization of a 'for' loop.
 - vi. [V1005](#). The 'ordered' directive is not present in an ordered loop.
 - vii. [V1006](#). Missing omp.h header file. Use '#include <omp.h>'.
 - viii. [V1101](#). Redefining number of threads in a parallel code.
 - ix. [V1102](#). Non-symmetrical use of set/unset functions for the following lock variable(s): foo.
 - x. [V1103](#). Threads number dependent code. The 'omp_get_num_threads' function is used in an arithmetic expression.
 - xi. [V1104](#). Redefining nested parallelism in a parallel code.
 - xii. [V1201](#). Concurrent usage of a shared resource via an unprotected call of the 'foo' function.
 - xiii. [V1202](#). The 'flush' directive should not be used for the 'foo' variable, because the variable has pointer type.
 - xiv. [V1203](#). Using the 'threadprivate' directive is dangerous, because it affects the entire file. Use local variables or specify access type for each parallel block explicitly instead.
 - xv. [V1204](#). Data race risk. Unprotected static variable declaration in a parallel code.
 - xvi. [V1205](#). Data race risk. Unprotected concurrent operation with the 'foo' variable.
 - xvii. [V1206](#). Data race risk. The value of the 'foo' variable can be changed concurrently via the 'bar' function.
 - xviii. [V1207](#). Data race risk. The 'foo' object can be changed concurrently by a non-const function.
 - xix. [V1208](#). The 'foo' variable of reference type cannot be private.
 - xx. [V1209](#). Warning: The 'foo' variable of pointer type should not be private.
 - xxi. [V1210](#). The 'foo' variable is marked as lastprivate but is not changed in the last section.
 - xxii. [V1211](#). The use of 'flush' directive has no sense for private 'foo' variable, and can reduce performance.
 - xxiii. [V1212](#). Data race risk. When accessing the array 'foo' in a parallel loop, different indexes are used for writing and reading.
 - xxiv. [V1301](#). The 'throw' keyword cannot be used outside of a try..catch block in a parallel section.
 - xxv. [V1302](#). The 'new' operator cannot be used outside of a try..catch block in a parallel section.
 - xxvi. [V1303](#). The 'foo' function which throws an exception cannot be used in a parallel section outside of a try..catch block.
- e. Customer's Specific Requests
 - i. [V2001](#). Consider using the extended version of the 'foo' function here: bar.
 - ii. [V2002](#). Consider using the 'Ptr' version of the 'foo' function here: bar.
 - iii. [V2003](#). Explicit conversion from 'float/double' type to signed integer type.
 - iv. [V2004](#). Explicit conversion from 'float/double' type to unsigned integer type.
 - v. [V2005](#). C-style explicit type casting is utilized. Consider using: static_cast/const_cast/reinterpret_cast.
- 5. Additional information
 - a. [Credits and acknowledgements](#).

You can download full PVS-Studio documentation as single html file here: <http://www.viva64.com/en/d/full>

Overview

PVS-Studio is a static analyzer that detects errors in source code of C/C++/C++11 applications. There are 3 sets of rules included into PVS-Studio:

1. Diagnostics of 64-bit errors (Viva64)
2. Diagnostics of parallel errors (VivaMP)
3. Diagnostics of General-purpose errors

To learn more about each type of diagnosis, visit the corresponding pages. The PVS-Studio tool is intended for developers of contemporary applications and it integrates into the Visual Studio 2005/2008/2010 environment providing the programmer with a convenient user interface to analyze files, navigate through code and get reference information. You do not need to study documentation and settings preliminarily to work with the analyzer. The analyzer is ready to work right after it is installed.

The methodology of static code analysis we employ has significant advantages over other types of analysis since it allows you to cover the whole program code. The procedure of code check cannot damage the code itself in any way. The analysis process is completely controlled by person and it is the programmer who decides if it needs modification.

The PVS-Studio tool is an own development of the Russian company OOO "Program Verification Systems".

PVS-Studio's features:

- Integration with Visual Studio 2005/2008/2010;
- online-help;
- pdf-documentation;
- saving and loading of analysis results;
- capability of command line launch;
- support of all cores and processors;
- estimate of complexity of 64-bit code migration;
- support of Windows (LLP64) and Linux (LP64) data models;
- interactive filters;
- convenient integration into the team development process;
- marking of program text with the purpose of checking new code only.

PVS-Studio code analyzer is necessary if you:

- Develop new 64-bit applications;
- Carry out 32-bit code migration to 64-bit systems;
- Add support of parallel execution to your program with the help of OpenMP technology.

Getting acquainted with PVS-Studio source code analyzer

Abstract

In the article, a brief description of PVS-Studio source code analyzer is given.

Introduction

PVS-Studio is a static code analyzer by OOO "Program Verification Systems" designed for developers of modern resource-intensive applications. PVS-Studio combines the possibilities of 64-bit code analysis from Viva64 unit, parallel code analysis from VivaMP unit and general purpose analysis. It lets you develop, test, perform migration and verification and of course create applications in C/C++/C++11 at a high level of reliability.

OOO "Program Verification Systems" Company is engaged in development and sales of code analyzers. Our products are: Viva64, code analyzer for migration and development of 64-bit applications, and VivaMP, code analyzer for verification of parallel OpenMP programs. These analyzers proved to be in demand by one group of users. That is why we have created a new software product, PVS-Studio, which comprises the two tools and provides an integrated solution for development of modern resource-intensive applications in C/C++/C++11 languages. A new general purpose analysis rules set had also been added to PVS-Studio analyzer.

Features

PVS-Studio installation is quite simple. At your machine, Microsoft Visual Studio 2005/2008/2010 IDE should be installed. For 64-bit applications analysis, it is advisable to have a 64-bit compiler which is a part of Visual Studio.

After installation, PVS-Studio is integrated in Visual Studio menu as shown in figure.

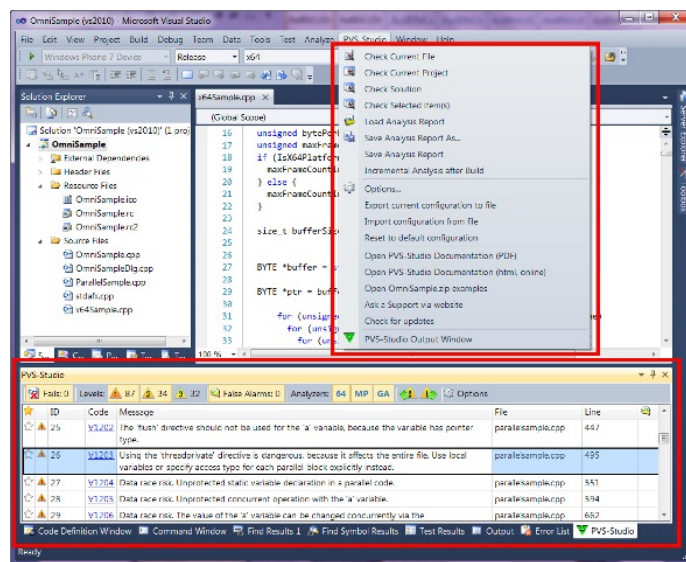


Figure 1: PVS-Studio integration into Microsoft Visual Studio

There are 3 sets of rules included into PVS-Studio:

1. Diagnostics of 64-bit errors (Viva64)
2. Diagnostics of parallel errors (VivaMP)
3. Diagnostics of general-purpose issues

PVS-Studio lets you detect the following bug types in the source code of programs in C/C++/C++11 languages:

- Errors of 32-bit applications migration to 64-bit systems;
- Errors occurring during new 64-bit application development;
- Non-optimal use of memory in 64-bit programs due to alignment peculiarities;
- Errors in parallel programs connected with lack of knowledge of OpenMP technology syntax;
- Errors in parallel programs connected with lack of knowledge of code paralleling laws using OpenMP;
- Errors occurring due to incorrect work with memory in parallel code (unguarded access to common memory, lack of synchronization, incorrect mode of access to variables, etc.).

All these groups of bugs occur both in new applications and in old ones either during attempts of porting them to a 64-bit platform or during code paralleling.

Using PVS-Studio analyzer, you can enhance the quality of a software product, reduce time of development and testing of the solution, and to provide code security.

Bugs search is conducted by static analysis technology, this lets you diagnose problems without running the application and the analysis is independent from your development environment. This is particularly important for errors diagnostics in parallel programs.

All the diagnosed errors are described in detail in the online help system that can be opened from PVS-Studio by clicking on the error message. [PVS-Studio help system](#) is also available online at our site.

In the distribution kit, special project examples of code bugs are supplied together with PVS-Studio, the analyzer work can be studied on these examples.

Conclusions

PVS-Studio code analyzer is necessary if you:

- Develop new 64-bit applications;
- Carry out 32-bit code migration to 64-bit systems;

- Add support of parallel execution to your program with the help of OpenMP technology.

System requirements for PVS-Studio analyzer

The PVS-Studio analyzer is intended to work on the Windows platform. It integrates into Microsoft Visual Studio 2005/2008/2010 development environments. System requirements for the analyzer coincide with requirements for Microsoft Visual Studio:

- Development environment: Microsoft Visual Studio 2005/2008/2010. It is advisable to install the "X64 Compilers and Tools" Visual Studio component for the analysis of 64-bit applications. It is included into all the mentioned versions of Visual Studio and can be installed through Visual Studio Setup. Note that PVS-Studio cannot work with Visual C++ Express Edition since this system does not support extension packages.
- Operating system: Windows XP/2003/Vista/2008/7 x86 or x64. Your operating system does not necessarily need to be 64-bit to analyze 64-bit applications.
- Hardware: PVS-Studio works on systems with main memory of 1 GB at least (the recommended size is 2 GBs or more); the analyzer supports multi-core operation (the more cores you have, the faster code analysis is).

Registration

To learn more about ordering and conditions of PVS-Studio licensing visit the [order page](#) on the site www.viva64.com.

We have invented a concept of "clicks" - passing to the source code containing errors through clicking in the message list. [Having downloaded PVS-Studio](#) without any registration, a potential user gets a full-blown version of the code analyzer. They can use it to check any projects as many times as needed. They get the whole message list. But they have a limited number of clicks (transfers to code). For example, it can be 100. But the number is not crucial and perhaps we will change it. Thus, a user can look through and pass to 100 diagnostics. No limitations, mind it.

If the user has exhausted the number of clicks, there is one of the following decisions for him/her to make:

- purchase a license;
- refuse using the tool if he/she doesn't like it;
- ask us to prolong the trial mode and provide personal information to us so that we can communicate with this user further via e-mail.

When purchasing a license, the user gets a usual registration key for one year and can use the tool at full extent (no clicks to worry about). When user refuses to use the tool, everything is clear too.

But what prolongation of the trial mode is concerned, it is implemented in PVS-Studio 4.54 version in the following way. When the number of clicks is exhausted by user, he/she sends us an e-mail with the following information: name, company, reason for prolonging the trial mode. And we give him/her a key for one more week. That is, there is some amount of manual work on our side.

We will change it in the next version after 4.54. The program will have an automatic form to fill in this information and after sending it the user will get more 500 clicks, for instance. The trial mode can be prolonged only once.

If the user doesn't want to prolong trial, he/she will have the following restrictions:

1. When checking new projects no names of files containing errors will be shown. Instead, there will be the phrase "TRIAL RESTRICTION".
2. If user opens an already saved log with found errors, transfer to the code by click won't work.

Of course, even when user has 0 clicks, he/she can open a preliminarily saved report and perform navigation manually, i.e. open a file and go to the corresponding line there. You can do it. But you should understand that static analysis is a tool that, first of all, allows you to save time (due to early detection of errors in software before its release, not after). But if our potential customer's time is so cheap or even free that he/she settles for manual navigation, static analysis is not for him/her and such a person is not our customer anyway.

PVS-Studio Release History

[PVS-Studio 4.54 \(February 1, 2012\)](#)
[PVS-Studio 4.53 \(January 19, 2012\)](#)
[PVS-Studio 4.52 \(December 28, 2011\)](#)
[PVS-Studio 4.51 \(December 22, 2011\)](#)
[PVS-Studio 4.50 \(December 15, 2011\)](#)
[PVS-Studio 4.39 \(November 25, 2011\)](#)
[PVS-Studio 4.38 \(October 12, 2011\)](#)
[PVS-Studio 4.37 \(September 20, 2011\)](#)
[PVS-Studio 4.36 \(August 31, 2011\)](#)
[PVS-Studio 4.35 \(August 12, 2011\)](#)
[PVS-Studio 4.34 \(July 29, 2011\)](#)
[PVS-Studio 4.33 \(July 21, 2011\)](#)
[PVS-Studio 4.32 \(July 15, 2011\)](#)
[PVS-Studio 4.31 \(July 6, 2011\)](#)
[PVS-Studio 4.30 \(June 23, 2011\)](#)
[PVS-Studio 4.21 \(May 20, 2011\)](#)
[PVS-Studio 4.20 \(April 29, 2011\)](#)
[PVS-Studio 4.17 \(April 15, 2011\)](#)
[PVS-Studio 4.16 \(April 1, 2011\)](#)
[PVS-Studio 4.15 \(March 17, 2011\)](#)
[PVS-Studio 4.14 \(March 2, 2011\)](#)
[PVS-Studio 4.13 \(February 11, 2011\)](#)
[PVS-Studio 4.12 \(February 7, 2011\)](#)
[PVS-Studio 4.11 \(January 28, 2011\)](#)
[PVS-Studio 4.10 \(January 17, 2011\)](#)
[PVS-Studio 4.00 \(December 24, 2010\)](#)

PVS-Studio 4.54 (February 1, 2012)

- New trial mode was implemented. As of now only a total number of clicks on messages will be limited. More details can be found in our blog or documentation.
- New menu command "Disable Incremental Analysis until IDE restart" was added. Sometimes disabling the incremental analysis can be convenient, for instance when editing some core h-files, as it forces a large number of files to be recompiled. But it should not be disabled permanently, only temporary, as one can easily forget to turn it on again later. This command is also available in the system tray during incremental analysis.
- New diagnostic [V602](#). Consider inspecting this expression. '<' possibly should be replaced with '<<'.
- New diagnostic [V603](#). The object was created but it is not being used. If you wish to call constructor, 'this->Foo::Foo(...)' should be used.
- New diagnostic [V807](#). Decreased performance. Consider creating a pointer/reference to avoid using the same expression repeatedly.
- New article in documentation: "[PVS-Studio menu commands](#)".

PVS-Studio 4.53 (January 19, 2012)

- New command for team work: "[Add TODO comment for Task List](#)". PVS-Studio allows you to automatically generate the special TODO comment containing all the information required to analyze the code fragment marked by it, and to insert it into the source code. Such comment will immediately appear inside the Visual Studio Task List window.
- New diagnostic [V599](#). The virtual destructor is not present, although the 'Foo' class contains virtual functions.
- New diagnostic [V600](#). Consider inspecting the condition. The 'Foo' pointer is always not equal to NULL.
- New diagnostic [V601](#). An odd implicit type casting.

PVS-Studio 4.52 (December 28, 2011)

- Changes were introduced to the .sln-file independent analyzer command line mode. It is now possible to start the analysis in several processes simultaneously, the output file (--output-file) will not be lost. The entire command line of arguments including the filename should be passed into the cl-params argument: --cl-params \$(CFLAGS) \$**.
- The "Analysis aborted by timeout" error was fixed, it could have been encountered while checking .sln file through PVS-Studio.exe command line mode.
- New diagnostic [V597](#). The compiler could delete the 'memset' function call, which is used to flush 'Foo' buffer. The RtlSecureZeroMemory() function should be used to erase the private data.
- New diagnostic [V598](#). The 'memset/memcpy' function is used to nullify/copy the fields of 'Foo' class. Virtual method table will be damaged by this.

PVS-Studio 4.51 (December 22, 2011)

- The issue concerning the #import directive when using Clang preprocessor was fixed. #import is supported by Clang differently from Microsoft Visual C++, therefore it is impossible to use Clang with such files. This directive is now automatically detected, and Visual C++ preprocessor is used for these files.
- "[Don't Check Files](#)" settings used for file and directory exclusions were significantly revised. As of now the folders to be excluded (either by their full and relative paths or by a mask) could be specified independently, as well as the files to be excluded (by their name, extension or a mask as well).
- Some libraries were added to the default exclusion paths. This can be modified on the "[Don't Check Files](#)" page.

PVS-Studio 4.50 (December 15, 2011)

- An external preprocessor is being utilized to preprocess files with PVS-Studio. It is only Microsoft Visual C++ preprocessor that had been employed for this task in the past. But in 4.50 version of PVS-Studio the support for the Clang preprocessor had been added, as its performance is significantly higher and it lacks some of the Microsoft's preprocessor shortcomings (although it also possesses issues of its own). Still, the utilization of Clang preprocessor provides an increase of operational performance by 1.5-1.7 times in most cases. However there is an aspect that should be considered. The preprocessor to be used can be specified from within the PVS-Studio Options -> Common Analyzer Settings -> Preprocessor field. The available options are: VisualCPP, Clang and VisualCPPAfterClang. The first two of these are self evident. The third one indicates that Clang will be used at first, and if preprocessing errors are encountered, the same file will be preprocessed by the Visual C++ preprocessor instead. This option is a default one (VisualCPPAfterClang).
- By default the analyzer will not produce diagnostic messages for libpng and zlib libraries (it is still possible to re-enable them).
- New diagnostic [V596](#). The object was created but it is not being used. The 'throw' keyword could be missing.

PVS-Studio 4.39 (November 25, 2011)

- New diagnostics were implemented (V594, V595).
- By default the analyzer will not produce diagnostic messages for Boost library (it is still possible to re-enable them).
- Progress dialog will not be shown anymore during incremental analysis, an animated tray icon, which itself will allow pausing or aborting the analysis, will be used instead.
- New "Don't Check Files and hide all messages from ..." command was added to the output window context menu. This command allows you to filter the messages and afterwards prevent the verification of files from the specified directories. The list of filtered directories can be reviewed in "Don't Check Files" options page.
- The detection of Intel C++ Compiler integration have been revamped – PVS-Studio will not run on projects using this compiler, it is required to replace the compiler with Visual C++ one.
- "Quick Filters" functionality was implemented. It allows filtering all the messages which do not meet the specified filtering settings.

PVS-Studio 4.38 (October 12, 2011)

- Speed increase (up to 25% for quad core computers).
- "Navigate to ID" command added to the context menu of PVS-Studio window.
- New "Find in PVS-Studio Output" tool window allows searching of keywords in analysis results.
- New diagnostic rules added (V2005).
- Options button on PVS-Studio Output Window was renamed to Suppression and now contain only three tab pages.

PVS-Studio 4.37 (September 20, 2011)

- New diagnostic rules added (V008, V2003, V2004).
- Now you can export PVS-Studio analysis report to text file.
- We use extended build number in some case.

PVS-Studio 4.36 (August 31, 2011)

- New diagnostic rules added (V588, V589, V590, V591, V592, V593).
- Changes in PVS-Studio menu.

PVS-Studio 4.35 (August 12, 2011)

- New diagnostic rules added (V583, V584, V806, V585, V586, V587).

PVS-Studio 4.34 (July 29, 2011)

- Now 64-bit analysis disabled by default.
- Now Incremental Analysis enabled by default.
- Changes of behavior in trial mode.
- PVS_STUDIO predefined macro was added.
- Fixed problem with Incremental Analysis on localized versions of Visual Studio.
- Balloon notification and tray icon (after analysis finished) was added.
- New diagnostic rules added (V582).
- Changed image to display on the left side of the wizard in the Setup program.

PVS-Studio 4.33 (July 21, 2011)

- Incremental Analysis feature now available for all versions of Microsoft Visual Studio (2005/2008/2010).
- Speed increase (up to 20% for quad core computers).
- New diagnostic rules added (V127, V579, V580, V581).

PVS-Studio 4.32 (July 15, 2011)

- Changes in PVS-Studio's licensing policy.
- Dynamic balancing of CPU usage.
- Stop Analysis button work faster.

PVS-Studio 4.31 (July 6, 2011)

- Fixed problem related to interaction with other extensions (including Visual Assist).
- New diagnostic rules added (V577, V578, V805).

PVS-Studio 4.30 (June 23, 2011)

- The full-fledged support for analyzer's operation through command line was implemented. It is possible to verify independent files or sets of files launching the analyzer from Makefile. Also the analyzer's messages can be viewed not only on screen (for each file), but they also can be saved into single file, which later can be opened in Visual Studio and the regular processing of the analysis' results can be performed, complete with setting up error codes, message filters, code navigation, sorting etc. [Details](#).
- New important mode of operation: Incremental Analysis. As of this moment PVS-Studio can automatically launch the analysis of modified files which are required to be rebuilt using 'Build' command in Visual Studio. All of developers in a team can now detect issues in newly written code without the inconvenience of manually launching the source code analysis – it happens automatically. Incremental Analysis operates similar to Visual Studio IntelliSense. The feature is available only in Visual Studio 2010. [Details](#).
- "Check Selected Item(s)" command was added.
- Changes in starting "Check Solution" via command line. [Details](#).
- New diagnostic rules added (V576).

PVS-Studio 4.21 (May 20, 2011)

- New diagnostic rules added (V220, V573, V574, V575).
- TFS 2005/2008/2010 integration was added.

PVS-Studio 4.20 (April 29, 2011)

- New diagnostic rules added (V571, V572).
- Experimental support for ARMV4/ARMV4I platforms for Visual Studio 2005/2008 (Windows Mobile 5/6, PocketPC 2003, Smartphone 2003).
- New "Show License Expired Message" option.

PVS-Studio 4.17 (April 15, 2011)

- New diagnostic rules added (V007, V570, V804)
- Incorrect display of analysis time in some locales has been fixed.
- New "Analysis Timeout" option. This setting allows you to set the time limit, by reaching which the analysis of individual files will be aborted with V006 error, or to completely disable analysis termination by timeout.
- New "Save File After False Alarm Mark" option. It allows to save or not to save a file each time after marking it as False Alarm.
- New "Use Solution Folder As Initial" option. It defines the folder which is opened while saving the analysis results file.

PVS-Studio 4.16 (April 1, 2011)

- It is possible now to define a list of files to be analyzed while launching the tool from command line. This can be used, for example, to check only the files which were updated by a revision control system. [Details](#).
- "Check only Files Modified In" option has been added into tool's settings. This option allows you to define the time interval in which the presence of modifications in analyzed files will be controlled using "Date Modified" file attribute. In other words, this approach would allow for verification of "all files modified today". [Details](#).

PVS-Studio 4.15 (March 17, 2011)

- There are much fewer false alarms in 64-bit analysis.
- Changes in the interface of safe-type definition.
- The error of processing stdafx.h in some special cases is fixed.
- Handling of the report file was improved.
- The progress dialogue was improved: you can see the elapsed time and the remaining time.

PVS-Studio 4.14 (March 2, 2011)

- There are much fewer false alarms in 64-bit analysis.
- New diagnostic rules were added (V566, V567, V568, V569, V803).
- A new column "Asterisk" was added in the PVS-Studio message window - you may use it to mark interesting diagnoses with the asterisk to discuss them with your colleagues later. The marks are saved in the log file.
- Now you may access PVS-Studio options not only from the menu (in the usual settings dialogue) but in the PVS-Studio window as well. This makes the process of setting the tool quicker and more convenient.

- Now you may save and restore PVS-Studio settings. It enables you to transfer the settings between different computers and workplaces. We also added the "Default settings" command.
- The state of PVS-Studio window's buttons (enabled/disabled) is saved when you launch Microsoft Visual Studio for the next time.

PVS-Studio 4.13 (February 11, 2011)

- New diagnostic rules are added V563, V564, and V565).
- The "Check for updates" command is added into the PVS-Studio menu.
- The "Hide all VXXX errors" command is added into context menu in PVS-Studio window. If you wish to enable the display of VXXX error messages again you can do it through PVS-Studio->Options->Detectable errors page.
- Suppressing false positives located within macro statements (#define) is added.

PVS-Studio 4.12 (February 7, 2011)

- New diagnostic rules are added (V006, V204, V205, V559, V560, V561, and V562).
- Changes in V201 and V202 diagnostic rules.

PVS-Studio 4.11 (January 28, 2011)

- V401 rule changed to V802.
- Fixed bug with copying messages to clipboard.

PVS-Studio 4.10 (January 17, 2011)

- New diagnostic rules are added (V558).

PVS-Studio 4.00 (December 24, 2010)

- New diagnostic rules are added (V546-V557).
- The issue of processing property sheets in Visual Studio 2010 is fixed.
- The error of traversing projects' tree is fixed.
- The "Project" field is added into the PVS-Studio window – it shows the project the current diagnostic message refers to.
- The issue of installing PVS-Studio for Visual Studio 2010 is fixed - now PVS-Studio is installed not only for the current user but for all the users.
- The crash is fixed occurring when trying to save an empty report file.
- The issue of absent safe_types.txt file is fixed.
- The error is fixed which occurred when trying to check files included into the project but actually absent from the hard disk (for instance, autogenerated files).
- Indication of processing the project's tree is added.
- The file with PVS-Studio's analysis results (.plog extension) is now loaded by double-click.
- The licensing policy is changed.

PVS-Studio 4.00 BETA (November 24, 2010)

- A new set of general-purpose static analysis rules (V501-V545, V801).
- New diagnostic rules are added (V124-V126).
- Changes in the licensing policy.
- A new window for diagnostic messages generated by the analyzer.
- Speed increase.

Release history for old versions

Please read release history for old versions [here](#).

Old PVS-Studio Release History (before 4.00)

[PVS-Studio 3.64 \(27 September 2010\)](#)

[PVS-Studio 3.63 \(10 September 2010\)](#)

[PVS-Studio 3.62 \(16 August 2010\)](#)

[PVS-Studio 3.61 \(22 July 2010\)](#)

[PVS-Studio 3.60 \(10 June 2010\)](#)

[PVS-Studio 3.53 \(7 May 2010\)](#)

[PVS-Studio 3.52 \(27 April 2010\)](#)

[PVS-Studio 3.51 \(16 April 2010\)](#)

[PVS-Studio 3.50 \(26 March 2010\)](#)

[PVS-Studio 3.44 \(21 January 2010\)](#)

[PVS-Studio 3.43 \(28 December 2009\)](#)

[PVS-Studio 3.42 \(9 December 2009\)](#)

[PVS-Studio 3.41 \(30 November 2009\)](#)

[PVS-Studio 3.40 \(23 November 2009\)](#)

[PVS-Studio 3.30 \(25 September 2009\)](#)

[PVS-Studio 3.20 \(7 September 2009\)](#)

[PVS-Studio 3.10 \(10 August 2009\)](#)

[PVS-Studio 3.00 \(27 July 2009\)](#)

[VivaMP 1.10 \(20 April 2009\)](#)

[VivaMP 1.00 \(10 March 2009\)](#)

[VivaMP 1.00 beta \(27 November 2008\)](#)

[Viva64 2.30 \(20 April 2009\)](#)

[Viva64 2.22 \(10 March 2009\)](#)
[Viva64 2.21 \(27 November 2008\)](#)
[Viva64 2.20 \(15 October 2008\)](#)
[Viva64 2.10 \(05 September 2008\)](#)
[Viva64 2.0 \(09 July 2008\)](#)
[Viva64 1.80 \(03 February 2008\)](#)
[Viva64 1.70 \(20 December 2007\)](#)
[Viva64 1.60 \(28 August 2007\)](#)
[Viva64 1.50 \(15 May 2007\)](#)
[Viva64 1.40 \(1 May 2007\)](#)
[Viva64 1.30 \(17 March 2007\)](#)
[Viva64 1.20 \(26 January 2007\)](#)
[Viva64 1.10 \(16 January 2007\)](#)
[Viva64 1.00 \(31 December 2006\)](#)

Please read actual release history [here](#).

PVS-Studio 3.64 (27 September 2010)

- Major documentation update, new sections was added.

PVS-Studio 3.63 (10 September 2010)

- Fixed bug which occurred sometimes during analysis of files located on non-system partitions.
- Fixed bug in calculation of macros' values for certain individual files (and not the whole project).
- "What Is It?" feature was removed.
- Issues examples for 64-bit code (PortSample) and parallel code (ParallelSample) are merged into single OmniSample example, which is described particularly in documentation.
- Fixed crash related to presence of unloaded project in Visual Studio solution.

PVS-Studio 3.62 (16 August 2010)

- New rule V123: Allocation of memory by the pattern "(X*)malloc(sizeof(Y))"
- The analysis of the code from command line (without Visual Studio project) is improved.
- Diagnostic messages from tti/tlh files do not produced by default.

PVS-Studio 3.61 (22 July 2010)

- Fixed crash in VS2010 with EnableAllWarnings key enabled in project settings.
- Fixed bug related to analysis projects that does excluded from build in Configuration Manager.
- The analysis of the code is considerably improved.

PVS-Studio 3.60 (10 June 2010)

- New rule V122: Memsized type is used in the struct/class.
- New rule V303: The function is deprecated in the Win64 system. It is safer to use the NewFOO function.
- New rule V2001: Consider using the extended version of the FOO function here.
- New rule V2002: Consider using the 'Ptr' version of the FOO function here.

PVS-Studio 3.53 (7 May 2010)

- "What Is It?" feature is added. Now you can ask PVS-Studio developers about diagnostic messages produced by our analyzer.
- The analysis of the code related to usage of unnamed structures is considerably improved.
- Fixed bug in structure size evaluation in certain cases.

PVS-Studio 3.52 (27 April 2010)

- New online help has been added. The previous help system integrated into MSDN. It was not very convenient for some reasons (both for us and users). Now PVS-Studio will open the help system on our site. We refused to integrate it into MSDN anymore. As before, the pdf-version of the documentation is also available.
- We stopped supporting Windows 2000.
- The settings page "Exclude From Analysis" was deleted - there is now the page "Don't Check Files" instead.
- Work in Visual Studio 2010 was improved.
- We eliminated the issue of integration into VS2010 when reinstalling.
- We fixed work of the function "Mark As False Alarm" with read-only files.

PVS-Studio 3.51 (16 April 2010)

- PVS-Studio supports Visual Studio 2010 RTM.
- New rule: V003: Unrecognized error found...
- New rule: V121: Implicit conversion of the type of 'new' operator's argument to size_t type.
- You may specify filemasks on the tab "[Don't Check Files](#)" to exclude some files from analysis.
- "Exclude From Analysis" option page improved.
- MoreThan2Gb option removed from "[Viva64](#)" option page (this option is deprecated).
- If you want check code from command line then you must indicate analyzer type (Viva64 or VivaMP).
- Priority of analyzer's process is reduced. Now you can work on computer more suitable while analysis is running.

PVS-Studio 3.50 (26 March 2010)

- PVS-Studio supports Visual Studio 2010 RC. Although Visual Studio has not been released officially yet, we have already added the support for this environment into the analyzer. Now PVS-Studio integrates into Visual Studio 2010 and can analyze projects in this environment. Help system in Visual

Studio 2010 has been changed, so the Help section of PVS-Studio does not integrate into the documentation yet as it is done in Visual Studio 2005/2008. But you still may use online-Help. Support of Visual Studio 2010 RC is not complete.

- A new PDF-version of Help system is available. Now we ship a 50-page PDF-document in the PVS-Studio distribution kit. It is a full copy of our Help system (that integrates into MSDN in Visual Studio 2005/2008 and is available online).
- PVS-Studio now has a new mechanism that automatically checks for new versions of the tool on our site. Checking for the updates is managed through the new option CheckForNewVersions in the settings tab called "Common Analyzer Settings". If the option CheckForNewVersions is set to True, a special text file is downloaded from www.viva64.com site when you launch code testing (the commands Check Current File, Check Current Project, Check Solution in PVS-Studio menu). This file contains the number of the latest PVS-Studio version available on the site. If the version on the site is newer than the version installed on the user computer, the user will be asked for a permission to update the tool. If the user agrees, a special separate application PVS-Studio-Updater will be launched that will automatically download and install the new PVS-Studio distribution kit. If the option CheckForNewVersions is set to False, it will not check for the updates.
- We have implemented the support for the standard C++0x at the level it was done in Visual Studio 2010. Now it supports lambda expressions, auto, decltype, static_assert, nullptr, etc. In the future, as C++0x support in Visual C++ is developing, the analyzer PVS-Studio will also provide support for the new C++ language capabilities.
- Now you can check solutions with PVS-Studio from the command line instead of Visual Studio environment. Note that we still mean that the checking will be performed from Visual Studio involving the files of projects (.vcproj) and solutions (.sln) but it will be launched from the command line instead of IDE. This way of launching the tool may be useful when you need to regularly check the code with the help of build systems or continuous integration systems.
- New rule V1212: Data race risk. When accessing the array 'foo' in a parallel loop, different indexes are used for writing and reading.
- We added a code signature certificate in the new version of our tool. It is done for you to be sure that the distribution kit is authentic, and get fewer warnings from the operating system when installing the application.

PVS-Studio 3.44 (21 January 2010)

- Partial support of code testing for Itanium processors. Now the code that builds in Visual Studio Team System for Itanium processors may be also tested with the analyzer. Analysis can be performed on x86 and x64 systems but analysis on Itanium is not implemented yet.
- We reduced the number of the analyzer's false alarms when analyzing an array access. Now, in some cases, the analyzer "understands" the ranges of values in the for loop and does not generate unnecessary warnings on accessing arrays with these indexes. For example: `for (int i = 0; i < 8; i++) arr[i] = foo();` // no warning from the analyzer.
- The number of the analyzer's false alarms is reduced - we introduced a list of data types that do not form large arrays. For example, HWND, CButton. Users may compose their own type lists.
- The installer error is corrected that occurs when installing the program into a folder different than the folder by default.

PVS-Studio 3.43 (28 December 2009)

- Option ShowAllErrorsInString removed (now it always has the value true).
- New rule V120: Member operator[] of object 'foo' declared with 32-bit type argument, but called with memsize type argument.
- New rule V302: Member operator[] of 'foo' class has a 32-bit type argument. Use memsize-type here.
- Operator[] analysis enhanced.
- Error of long removal of the program in case of recurrent installation "over the program again" corrected.
- Fixed problem related to analysis files with "" character in filename.

PVS-Studio 3.42 (9 December 2009)

- Errors diagnostics with magic numbers enhanced. Now in a message about a problem, more information is given out; this allows to use filters in a more flexible way.
- Error during work with precompiled header files of special type corrected.
- Option DoTemplateInstantiate is now turned on by default.
- Error with preprocessor hang-up at large number of preprocessor messages corrected.
- Analysis of operator[] enhanced.

PVS-Studio 3.41 (30 November 2009)

- Error of same name files analysis during work on a multicore machine corrected.
- Error of incorrect diagnostics of some types of cast-expressions corrected.
- Parsing of overloaded functions in the analyzer improved considerably.
- Diagnostics of incorrect use of time_t type added.
- Processing of special parameters in the settings of Visual C++ project files added.

PVS-Studio 3.40 (23 November 2009)

- A new feature "Mark as False Alarm" has been added. Due to it, it is now possible to mark those lines in the source code in which false alarm of the code analyzer happens. After such marking, the analyzer will not output any diagnostic messages for such code any more. This allows to use the analyzer constantly and more conveniently in the process of software development for new code verification.
- Project Property Sheets support added, a procedure of easy-to-use Visual Studio projects setup.
- During the verification of parallel programs, the analyzer can walk the code twice, this will allow to collect more information and carry out more precise diagnostics of some errors.

PVS-Studio 3.30 (25 September 2009)

- In PVS-Studio, the possibility of testing 32-bit projects for estimating the complexity and cost of code migration to 64-bit systems has been added.
- A new rule for 64-bit code analysis has been added, V118: malloc() function accepts a dangerous expression in the capacity of an argument.
- A new rule for 64-bit code analysis has been added, V119: More than one sizeof() operators are used in one expression.
- A new rule for parallel code analysis has been added, V1211: The use of 'flush' directive has no sense for private '%1%' variable, and can reduce performance.
- Combined operation with Intel C++ Compiler has been improved (crash at the attempt of code verification with installed Intel C++ Compiler has been corrected.)
- Localized versions of Visual Studio support has been enhanced.

PVS-Studio 3.20 (7 September 2009)

- The error of incorrect output of some messages in Visual Studio localized versions has been corrected.
- Log-file loading improved.
- Critical errors processing improved - now it is easy to inform us on possible tools problems.
- Installer operation improved.
- Project files walking error corrected.

PVS-Studio 3.10 (10 August 2009)

- Templates instantiating support has been added. Now the search of potential errors is carried out not simply by template body (as it was earlier), but also template parameters substitution is made for more thorough diagnostics.

- The code analyzer can work in the mode of Linux environment simulation. We have added the support of various data models. That is why, now it is possible to verify cross platform programs on a Windows system the way it would be carried out on a Linux system.
- The error connected with incorrect functioning of the analyzer of parallel errors in 32-bit environment has been corrected.
- The work of the analyzer with templates has been considerably improved.

PVS-Studio 3.00 (27 July 2009)

- Software products Viva64 and VivaMP are united into one program complex PVS-Studio.
- The new version is a significantly upgraded software product.
- Operation of the unit of integration into Visual Studio is much more stable.
- Operation rate in multi-processor systems is increased: analysis is performed in several threads, and the number of the analyzer's operating threads can be set with the help of "Thread Count" option. By default the number of threads corresponds to the number of cores in the processor but it can be reduced.
- A possibility to operate the analyzer from the command line is added. A new option "Remove Intermediate Files" is added into the settings of the program which allows you not to remove command files created during the code analyzer's operation. These command files can be launched separately without launching Visual Studio to perform analysis. Besides, when creating new command files you can perform by analogy analysis of the whole project without using Visual Studio.
- It became more simple, convenient and quick to operate diagnosis of separate errors. Now you can enable and disable the function of showing separate errors in the analysis' results. What is the most important is that changing of the message list is performed automatically without the necessity of relaunching analysis. Having performed analysis you can scroll through the list of errors or simply disable showing of those errors which are not relevant to you project.
- Operating with error filters has been improved greatly. Filters for hiding some messages are now defined simply as a list of strings. Like in case of diagnosing separate errors, using filters doesn't demand relaunching analysis.
- Change of licensing policy. Although PVS-Studio is a single product, we provide licensing both for separate analysis units such as Viva64 and VivaMP and for all the units together. Besides, there are licenses for one user or for a team of developers. All these changes are reflected in registration keys.
- Support of localized versions of Visual Studio has been improved greatly.
- Help system for a new version of PVS-Studio integrating into MSDN has been modified and improved greatly. Description of new sections allows you to master operation with the software product better.
- Graphic design of the software product has been improved. New icons and graphics in the installer make the analyzer's appearance more beautiful.

VivaMP 1.10 (20 April 2009)

- The analysis of the code containing calls of the class static functions has been improved.
- New diagnostic rules for the analysis of errors connected with the exceptions V1301, V1302, V1303 have been implemented.
- The error of the incorrect display of the analysis progress indicator on machines with non-standard DPI has been corrected.
- Some other enhancements have been implemented.

VivaMP 1.00 (10 March 2009)

- VivaMP 1.00 release.

VivaMP 1.00 beta (27 November 2008)

- First public beta version release on the Internet.

Viva64 2.30 (20 April 2009)

- New diagnostic rule V401 has been implemented.
- Constants processing has been improved, in a number of cases, this reduces the quantity of false diagnostic warnings.
- The error of the incorrect display of the analysis progress indicator on machines with non-standard DPI has been corrected.
- A number of errors have been corrected.

Viva64 2.22 (10 March 2009)

- Collaboration of Viva64 and VivaMP is improved.
- Analyzer performance is improved up to 10%.

Viva64 2.21 (27 November 2008)

- Collaboration of Viva64 and VivaMP is added.

Viva64 2.20 (15 October 2008)

- Diagnosis of potentially unsafe constructions is improved. As the result the number of the code analyzer's "false alarms" is reduced approximately by 20%. Now the developer will spend less time to analyze the code diagnosed as potentially unsafe.
- Help system is amended. It has been extended and new examples have been added. As diagnosis of potentially unsafe constructions is improved in this version Help system has been also supplemented with explanations concerning the constructions which are now considered safe.
- The speed of a project's structure analysis is raised. Now the same work is performed 10 times quicker. As the result the total time of the whole project's analysis is reduced.
- C++ template analysis is improved. It's not a secret that far not all the code analyzers understand templates. We're constantly working to improve diagnosis of potentially unsafe constructions in templates. Such an improvement is made in this version.
- Format of some code analyzer's messages is amended to make it possible to set filters more accurately. Thus now, for example, the analyzer doesn't only inform about an incorrect index type while accessing an array but also shows the name of the array itself. If the developer is sure that such an array cannot cause problems in 64-bit mode at all he can filter all the messages concerning this array's name.

Viva64 2.10 (05 September 2008)

- Visual C++ 2008 Service Pack 1 support is added.

Viva64 2.0 (09 July 2008)

- Visual C++ 2008 Feature Pack (and TR1) support is added.
- Pedantic mode is added which allows you to find constructions potentially dangerous but rarely causing errors.
- Diagnosis of template functions is improved

Viva64 1.80 (03 February 2008)

- Visual Studio 2008 is fully supported now.
- Source code analysis speed is increased.
- Installer is improved. Now you can install Viva64 without administrator privileges for personal usage.

Viva64 1.70 (20 December 2007)

- The support of a new diagnostic message (V117) is added. Memsize type used in union.
- Fixed critical bug related to detection of more than one errors in source line.
- Fixed bug in type evaluation in some complex syntax.
- User Interface is improved. Now you can see a common analysis progress indicator.
- Visual Studio 2008 support is added (BETA).

Viva64 1.60 (28 August 2007)

- The support of a new diagnostic message (V112) is added. Dangerous magic number used.
- The support of a new diagnostic message (V115) is added. Memsize type used for throw.
- The support of a new diagnostic message (V116) is added. Memsize type used for catch.
- The restriction of a trial version is changed. In each analyzed file the location of only some errors is shown.

Viva64 1.50 (15 May 2007)

- C source analysis is fully supported. Now C source code may be analyzed correctly.

Viva64 1.40 (1 May 2007)

- Message Suppression feature added. You can adjust filters on the [Message Suppression](#) page of the Viva64 settings to ignore some of the warning messages. For example, you can adjust filters to skip messages with particular error codes and messages including names of specific variables and functions.
- Ability to save/load analysis results added.
- Analysis results representation improved. The results are now displayed in the Visual Studio standard Error List window, just like the compiler messages.

Viva64 1.30 (17 March 2007)

- Representation of the process of the code analysis is improved. Unnecessary windows switching are removed, a general progress bar is created.
- Toolbar with Viva64 commands is added.
- The user now can point the analyzer if its program is using more than 2GB of RAM. On using less than 2GB some warning messages are disabled.
- The support of a new diagnostic message (V113) is added. Implicit type conversion from memsize to double type or vice versa.
- The support of a new diagnostic message (V114) is added. Dangerous explicit type pointer conversion.
- The support of a new diagnostic message (V203) is added. Explicit type conversion from memsize to double type or vice versa.

Viva64 1.20 (26 January 2007)

- Filtration of repeating error messages is added. It is useful when there are errors in header files. Earlier if *.h file with an error included into different *.cpp files the warning message about the error in the *.h file was shown several times. Now there is only one message about in the *.h file shown.
- Now Viva64 informs about the number of errors found after the code analysis. You can always see:
 - how much code is left to be checked;
 - how many errors are corrected already;
 - which modules contain the largest number of errors.
- Support of some hot keys is added. Now you can interrupt the analyzer's work with the help of Ctrl+Break. In case you want to check the current file just press Ctrl+Shift+F7.
- There are some errors of the analyzer's work corrected.

Viva64 1.10 (16 January 2007)

- With the help of the Viva64 analyzer itself we've prepared the 64-bit version of Viva64 at once! But you should not care about the choose of the right version during the installation. The installer will find out itself which version should be installed for your operation system.
- The support of a new rule is added. Now the parameters of the functions with the variable number of arguments are checked (V111-error code).
- There is no unnecessary diagnosis of the address to the array item with the help of enum values.
- There is no unnecessary diagnosis of the constructions of type `int a = sizeof(int)`.
- The Help System is improved.

Viva64 1.00 (31 December 2006)

- First public release on the Internet.

Limitation of the analyzer

Analyzer doesn't fully support diagnosis of errors while using some C/C++/C++11 constructions. It may cause false warning messages or absence of messages in some cases.

Analyzer doesn't fully support some language extensions implemented in Visual C++. Neither is there support of some aspects of modern C++ standard.

The analyzer does not work with files in Unicode format either, nor with files containing Unicode symbols in their paths.

Main limitations:

- Incomplete support of complex templates (for example, with partial specialization);
- Incomplete support of overloaded functions;
- Analysis of managed code is not implemented;
- `msclr` namespace is not supported;

We should mention that in practice these limitations don't influence the code analysis quality and you just should be aware of their existence.

Common information on working with the PVS-Studio analyzer

[Abstract](#)

[System requirements and installation of PVS-Studio](#)

[Introduction into PVS-Studio](#)

[Fixing errors](#)

[How to work with the list of diagnostic messages](#)

[Is it necessary to fix all the potential errors the analyzer informs about?](#)

Abstract

The article is a tutorial on working with the PVS-Studio code analyzer. This section contains examples of how to perform most common tasks when working with the PVS-Studio analyzer.

System requirements and installation of PVS-Studio

The PVS-Studio analyzer is intended to work on the Windows platform. It integrates into Microsoft Visual Studio 2005/2008/2010 development environments. You may learn about the [system requirements](#) for the analyzer in the corresponding section of the documentation.

After you obtain the PVS-Studio installation package, you may start installing the program.



Figure 1 - Installation of PVS-Studio

The code analyzer will integrate into the Microsoft Visual Studio development environment during the installation process. In case you have several versions of Microsoft Visual Studio installed on your computer, the analyzer will integrate into them all automatically.

To make sure that the PVS-Studio tool was correctly installed, you may launch Microsoft Visual Studio and open the About Microsoft Visual Studio window (Help/About Microsoft Visual Studio). The PVS-Studio analyzer must be present in the list of installed components (Figure 2).

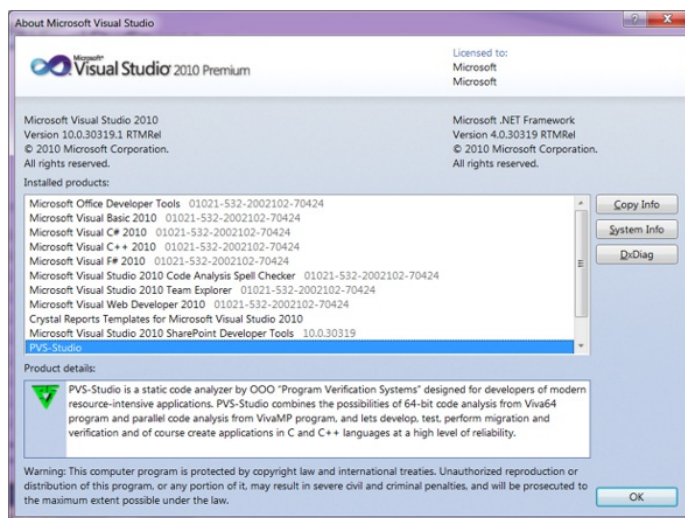


Figure 2 - About Microsoft Visual Studio window with the PVS-Studio component installed

Before you begin working in the program, we also recommend you to unpack the collection of samples of 64-bit and parallel errors called OmniSample into any folder you want from Start\PVS-Studio\ (64-bit and Parallel issues example, file OmniSample.zip)

With the help of this example you may study defects that in practice occur in contemporary software. OmniSample contains samples of issues occurring when porting software from 32-bit systems to 64-bit ones and also allow you to see what happens with parallel programs that have "parallel" errors. Further description in this article will be based on this sample collection.

Introduction into PVS-Studio

Let's open the OmniSample project (vs2010) in Microsoft Visual Studio 2010 (or in any other version of Microsoft Visual Studio you have).

There are several reasons why it is better to use the OmniSample project to get started with the PVS-Studio tool:

- It contains most of the code defects diagnosed by PVS-Studio Viva64 and VivaMP rulesets;
- It lets you study real-life behavior of an application with an error;
- the demo version of PVS-Studio shows positions of only several errors (although it detects all of them). But when you work with OmniSample, the program shows all the defects.

Let's see how PVS-Studio deals with 64-bit errors, for example. After opening the OmniSample project, choose the 64-bit configuration x64 to check for 64-bit code defects and launch analysis of the whole project by the "Check Solution" command as shown in Figure 3.

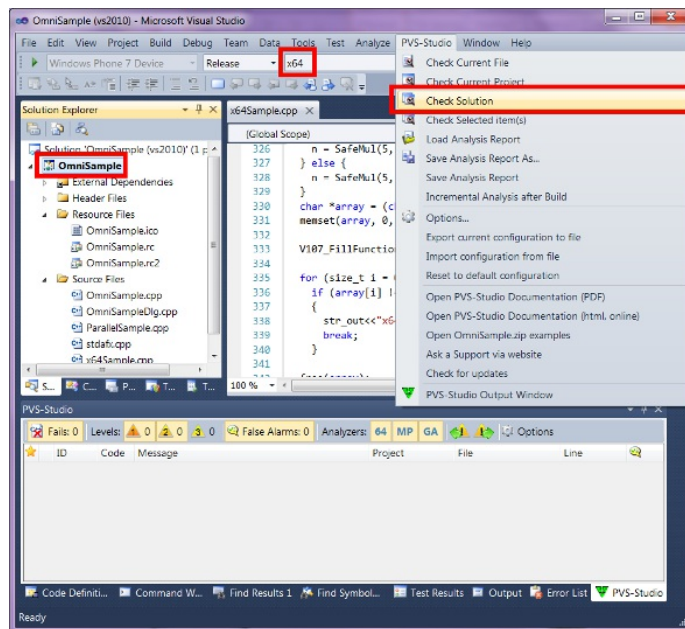


Figure 3 - Checking the OmniSample project for 64-bit issues

The x64 configuration is chosen deliberately because it is advisable to check 64-bit configurations for 64-bit issues. The point is that the project's settings in the 64-bit configuration differ from those in the 32-bit configuration, so it is not recommended to perform analysis of 64-bit code in the 32-bit configuration.

After launching the verification, the progress bar will appear with the buttons Pause (to pause the analysis) and Stop (to terminate the analysis). Potentially dangerous constructs will be displayed in the list of detected errors during the analysis procedure (Figure 4).

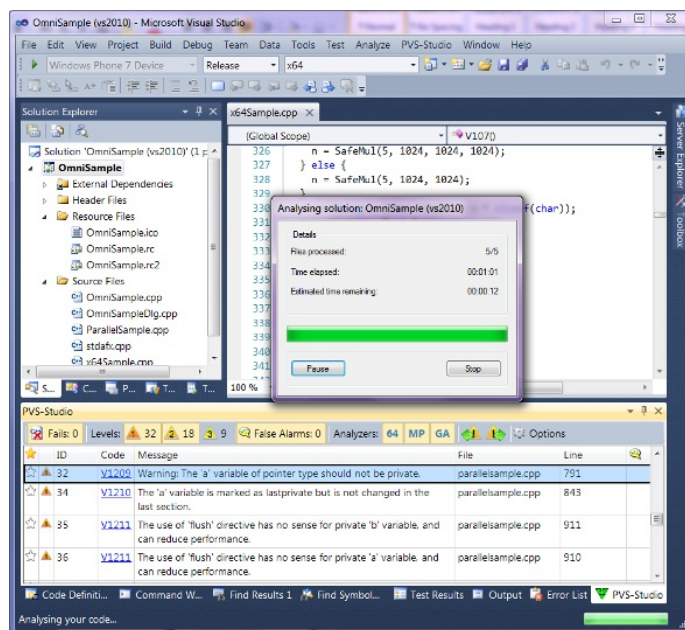


Figure 4 - Project analysis - issues detected in code are displayed in the window during the analysis

The term "a potentially dangerous construct" means that the analyzer considers a particular code line a defect. Whether this line is a real defect in an application or not is determined only by the programmer who knows the application. You must correctly understand this principle of working with code analyzers: no tool can completely replace a programmer when solving the task of fixing errors in programs. Only the programmer who relies on his knowledge can do this. But the tool can and must help him with it. That is why the main task of the code analyzer is to reduce the number of code fragments the programmer must look through and decide what to do with them.

Once the code analysis is over, you may look through the messages.

Fixing errors

After getting the list of diagnostic messages from the analyzer, you may study them. Let's look at the first error:

error V101: Implicit assignment type conversion to memsize type.
x64sample.cpp 24

Here is its code:

```
size_t bufferSize = imageWidth * imageHeight *
    bytePerPixel * maxFrameCountInBuffer;
```

The problem here is that the resulting buffer size (the bufferSize variable) has the right size `size_t` but the variables participating in the expression (imageWidth, imageHeight, bytePerPixel and maxFrameCountInBuffer) have the `int` type. Multiplication of these variables will result in a value that also has the `int` type. This is not crucial for values which are less than 2 Gbytes since there will be the type conversion. But if the result will be a number larger than 2 Gbytes, it will be "cut" to 2 Gbytes despite the fact that the bufferSize variable has the right type. To solve this issue, we need to change the types of the variables participating in the expression. We may do this in the code a bit above the mentioned fragment. Instead of:

```
unsigned imageWidth = 1000;
```

```

unsigned imageHeight = 1000;
unsigned bytePerPixel = 3;
unsigned maxFrameCountInBuffer;

```

We should write like this:

```

size_t imageWidth = 1000;
size_t imageHeight = 1000;
size_t bytePerPixel = 3;
size_t maxFrameCountInBuffer;

```

You may learn about this correction type in the help system. If you click on the cell containing the code of an error in the 'Code' column, you will see a window with the description for this error (Figure 5):

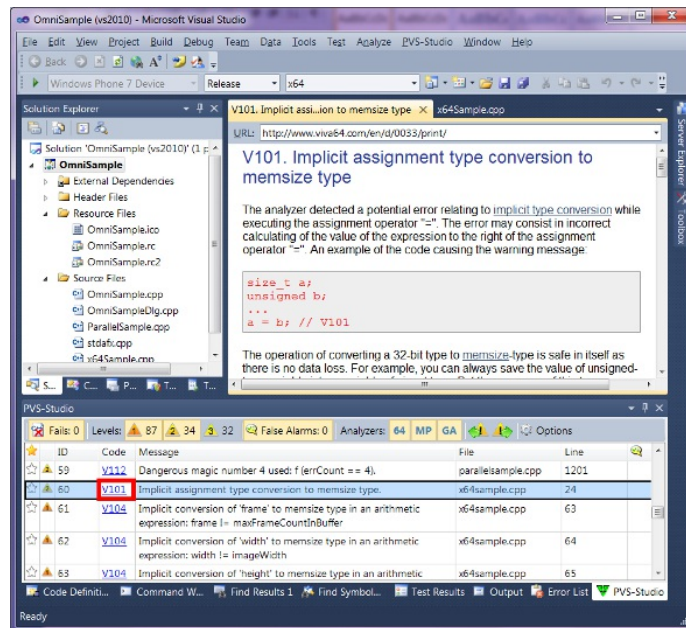


Figure 5 - Detailed description of an error and ways of fixing it

Having fixed the data types used here, let's restart the analysis to see that there are one less item in the diagnostic warnings. It means that the issue is fixed. In the same way we should review all the diagnostic messages and fix those fragments in the code where problems are possible.

How to work with the list of diagnostic messages

Of course, in real large projects, there will be not dozens but hundreds or even thousands of diagnostic messages and it will be a hard task to review them all. To make it easier, the PVS-Studio analyzer provides several mechanisms. The first mechanism is filtering by the error code. The second is filtering by the contents of the diagnostic messages' text. The third is filtering by file paths. Let's examine examples of using filtering systems.

Suppose you are sure that the diagnostic messages with the code V112 (using magic numbers) are never real errors in your application. In this case you may turn off the display of these diagnostic warnings in the analyzer's settings:

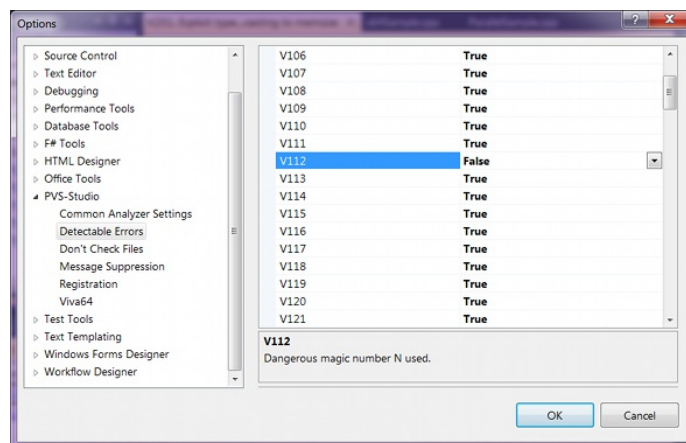


Figure 6 - Turning off some diagnostic messages by code

After that, all the diagnostic warnings with the code V112 will disappear from the error list. Note that you do not need to restart the analyzer. If you turn on these messages again, they will appear in the list without relaunching the analysis as well.

Now let's study another way of filtering by the text of diagnostic messages. Let's return to the OmniSample example. One of the errors in this sample is access to the array array using an index of the int type:

```

error V108: Incorrect index type: array[not a memsize-type].
Use memsize type instead. x64sample.cpp 390

```

This is the code:

```

volatile int index = 0;
for (size_t i = 0; i != n; ++i) {
    array[index++] = 1; // the problem is here
    if (array[i] != 1)
        throw CString("x64 portability issues");
}

```

Of course you may simply change the type of the index variable from unsigned to size_t and the problem will disappear. But if there are many code fragments like this with the access to the array variable and you know for sure that array has no more than two billion items, you may "ask" the analyzer not to show messages whose text contains the word "array". It is done in the settings on the MessageSuppression page:

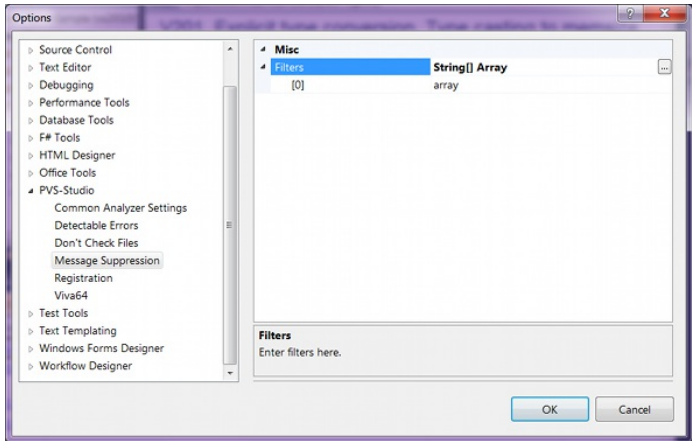


Figure 7 - Turning off some diagnostic messages by their text

After that, all the diagnostic messages whose text contains the word "array" will disappear from the error list without restarting the code analyzer. You may get them back by simply deleting the word "array" from the filter.

The last mechanism of reducing the number of diagnostic messages is filtering by masks of project files' names and file paths.

Suppose your project employs the Boost library. The analyzer will certainly inform you about potential issues in this library. But if you are sure that these messages are not relevant for your project, you may simply add the path to the folder with Boost on the page Don't check files (Figure 8):

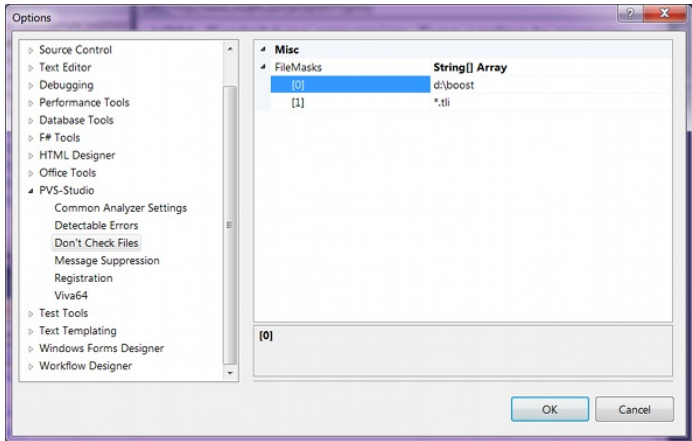


Figure 8 - Setting message filtering by file location and names

After that, the diagnostic messages referring to the files in this folder will not be shown. This option requires restarting the analysis.

Also, PVS-Studio has the "Mark as False Alarm" function. It enables you to mark those lines in your source code which cause the analyzer to generate false alarms. After marking the code, the analyzer will not produce diagnostic warnings on this code. This function makes it more convenient to use the analyzer permanently during the software development process when verifying newly written code.

Thus, in the following example, we turned off the diagnostic messages with the code V104:

```
size_t n = 100;
for (unsigned i = 0;
    i < n; //-V104
    i++)
{
    // ...
}
```

This function is described in more details in the section "[Suppression of false alarms](#)".

There are also some other methods to influence the display of diagnostic messages by changing the code analyzer's settings but they are beyond the scope of this article. We recommend you to refer to the documentation on the code analyzer's settings.

Is it necessary to fix all the potential errors the analyzer informs about?

When you have reviewed all the messages generated by the code analyzer, you will find both real errors and constructs which are not errors. The point is that the analyzer cannot detect 100% exactly all the errors in programs without producing the so called "false alarms". Only the programmer who knows and understands the program can determine if there is an error in each particular case. The code analyzer just significantly reduces the number of code fragments the developer needs to review.

So, there is certainly no reason for correcting all the potential issues the code analyzer refers to.

But you must attempt to fix as many fragments as possible. It is especially relevant when the static analyzer is used to verify an application not once, for instance, when you port it to a 64-bit system, but regularly with the purpose to find new errors and inefficient constructs brought into a project. In this case, correcting fragments which are really not errors and setting the analyzer for suppressing particular types of errors will significantly reduce time for the next launch of the analyzer.

Working Mode

[Analyzer's diagnostics units](#)

[Grouping analyzer's messages by their significance levels](#)

Analyzer's diagnostics units

PVS-Studio consists of several units used for analysis. At present they are:

- the unit for diagnosing problems in 64-bit code (Viva64);
- the unit for diagnosing problems in parallel OpenMP code (VivaMP).
- The unit for general analysis diagnostics;

To enable/disable the display of diagnostic messages belonging to one particular analyzer unit you can use special check buttons (64, MP, GA), as shown below (Figure 1):

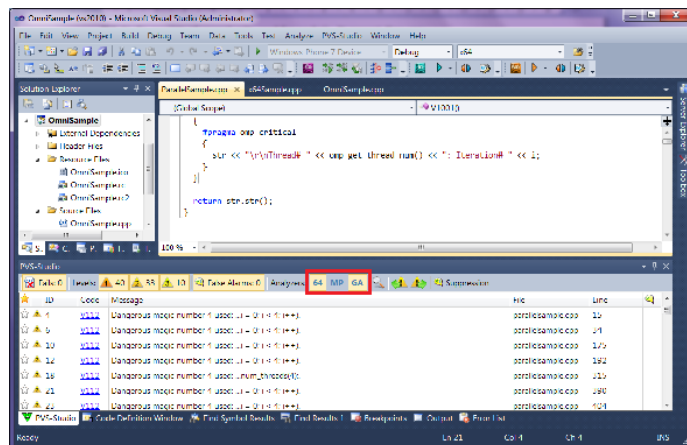


Figure 1 — PVS-Studio diagnostics units

It is not necessary to restart the analysis anew after changing the display settings for diagnostics units.

Grouping analyzer's messages by their significance levels

All messages produced by the analyzer are distributed among 4 groups: the 'Fails' group and the 3 significance levels – Level1, Level 2 and Level 3, as you can see below (figure 2):

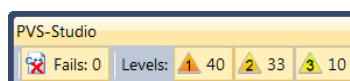


Figure 2 — Analyzer's messages being distributed among significance groups

The 'Fails' group contains the messages related to analyzer's operational errors (for instance the V001, V003 messages etc.), and also any unprocessed output produced by auxiliary utilities employed by the analyzer (preprocessor, cmd command line processor), which they themselves pass into stdout/stderr. For example, the 'Fails' group can contain preprocessor's source code compilation error, file access errors (the file was not found or it was blocked by antiviral software) etc.

All analyzer's diagnostics messages containing the potential issues in the source code are distributed among significance level groups. The importance of any of the messages produced by the static analyzer can be estimated from 2 distinctive parameters: criticality of the potential issue reported and the rate of false positives generation with it. According to these criteria each diagnostics message produced by the analyzer is assigned to one out of three significance levels. In this way, the Level 1 contains the most critical diagnostics which have the highest probability of being the real issues in the source code, and the Level 3 — the low-critical diagnostics or diagnostics with the highest false-positives generation. It's worth noting that the error's code not necessarily completely ties it to a certain significance level. The distribution of messages among these level groups is highly dependent of the context inside which they were generated.

Initially the third and second levels are disabled by default. To enable them you should use the corresponding check buttons, as was shown in figure 2.

Suppression of false alarms

[Abstract](#)

[Suppression of individual false positives.](#)

[Suppression of multiple false positives by using the group filtering mechanism](#)

[Demonstration of the Mark as False Alarm function using OmniSample project as example](#)

[Implementation of the false alarm suppression function](#)

[Suppressing false positives located within macro statements \(#define\)](#)

[Mass suppression of false positive alarms](#)

[Other means of filtering messages in the PVS-Studio analyzer](#)

[Possible issues](#)

Abstract

The article describes the function "Mark as False Alarm" that appeared in PVS-Studio 3.40 and provides an example of how to use it. This function allows you to mark those PVS-Studio analyzer diagnostic messages which are "false alarms" so that you would not have to see them at the next launch of the analyzer.

Suppression of individual false positives.

Any code analyzer always produces a lot of the so called "false alarms" besides helpful messages. These are situations when it is absolutely obvious to the programmer that the code does not have an error but it is not obvious to the analyzer. Such messages are called false alarms. Consider a sample of code:

```
ptrdiff_t value;
fread(&value, 4, 1, f);
char RGBA[4];
```

There will be two V112 warnings generated for this code since the magic constant 4 is used here. In the first case, it is an error because the size of a variable of the `ptrdiff_t` type will not equal four bytes in the 64-bit system. In the second case, number 4 signifies the number of color components and is safe. In PVS-Studio, beginning with the version 3.40, we have implemented the capability to mark an error message generated by PVS-Studio as a false alarm. You may do this either manually or with the help of the corresponding context menu command. Appearance of the "Mark as False Alarm" option in PVS-Studio greatly extends the potential of integrating the code analyzer into the software development process at the stage of everyday permanent use, which allows you not only port applications to the 64-bit platform but also make sure that there are no dangerous issues in new code you have just developed.

To suppress a false alarm, you may add a special comment into the code:

```
char RGBA[4]; //-V112
```

Now the analyzer will not generate the V112 warning on this line.

You may type the comment suppressing warnings into the code by yourself. You may also use a special command provided by PVS-Studio. The user is provided with two commands available from the PVS-Studio's context menu (see Figure 1).

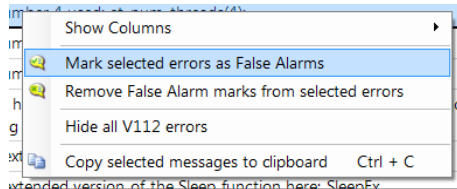


Figure 1 - Commands to work with the mechanism of false alarm suppression

Let's study the available commands concerning False Alarm suppression:

1. Mark selected errors as False Alarm. You may choose one or more false alarms in the list (see Figure 2) and use this command to mark the corresponding code as safe.

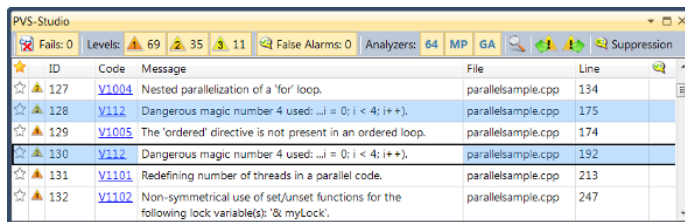


Figure 2 - Choosing warnings before executing the Mark Selected errors as False Alarms command

2. Remove False Alarm marks from selected errors. This command removes the comment that marks code as safe. This function might be helpful if, for instance, you were in a hurry and marked some code fragment as safe by mistake. Like in the previous case, you must choose the required messages from the list.

Suppression of multiple false positives by using the group filtering mechanism

It is possible that certain kinds of diagnostics are not essential for the project being analyzed (For example, if you are not interested in the errors relating to explicit type casting — V201, V202, V203 codes, e.t.c.), or one of the diagnostics produces warnings for the source code which, you have no doubt in it, is correct. In such a situation one could utilize the group suppression mechanism, which is based on filtering the analysis output results. The list of available filtering modes can be accessed through the "Suppressions" toolbar button, or through the common PVS-Studio -> Options menu (see figure 3)

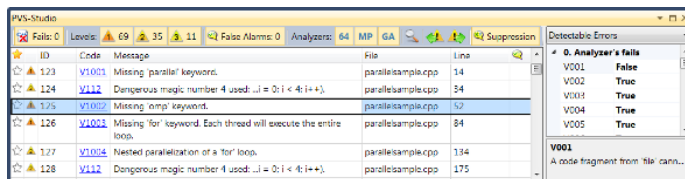


Figure 3 - The group message filtering modes

The group filtering modes include [Detectable Errors](#), [Don't Check Files](#) and [Message Suppression](#).

Utilizing the "Hide all Vxxx errors" context menu command (see in figure 1) it is possible to disable the display of all the errors belonging to a certain code. To enable the display of these errors again you should select the "Detectable Errors" mode and check the required code as True.

The suppression of multiple messages through filters does not require restarting of the analysis, the filtering results will appear in PVS-Studio output window immediately.

Demonstration of the Mark as False Alarm function using OmniSample project as example

Let's show how to use the Mark as False Alarm function with the demo OmniSample project, which is included into the PVS-Studio distribution package, as example.

At first you need to unpack OmniSample into any folder you like from menu Start\PVS-Studio\ (64-bit and Parallel issues example, file OmniSample.zip) or through Visual Studio 'PVS-Studio\Open Omnisample.zip examples' menu item and open it in the Microsoft Visual Studio environment.

After opening the project, let's launch analysis of the solution using the "Check Solution" command (Figure 4).

This comment informs the code analyzer that it must not generate the message about this error in this line when analyzing the project next time.

You may add this comment manually as well without using the "Mark selected errors as False Alarms" command, but you must follow the note's format: two slashes, minus (without a space), error code.

After marking the message as a false alarm, you may use the "False Alarms" check-button (Figure 7) to refresh the list of error messages and hide the unnecessary message. After that the PVS-Studio window will contain one less item.

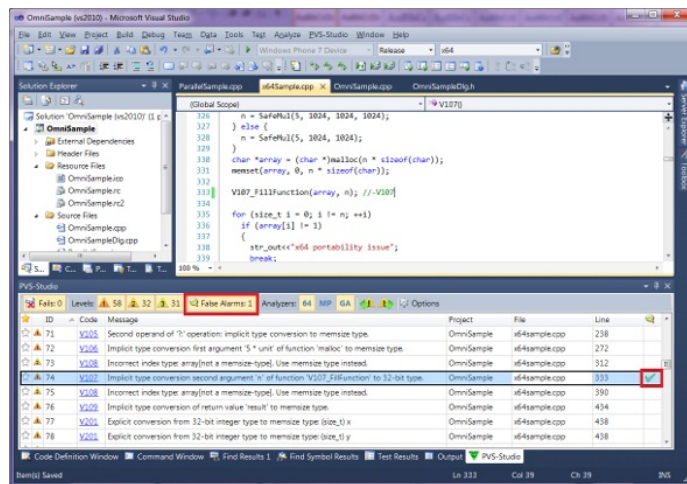


Figure 7 – The FA button controls the display of marked messages

You may remove the comment using the "Remove False Alarm marks from selected errors" command, having chosen the error message in the PVS-Studio window beforehand. You may also remove the comment manually.

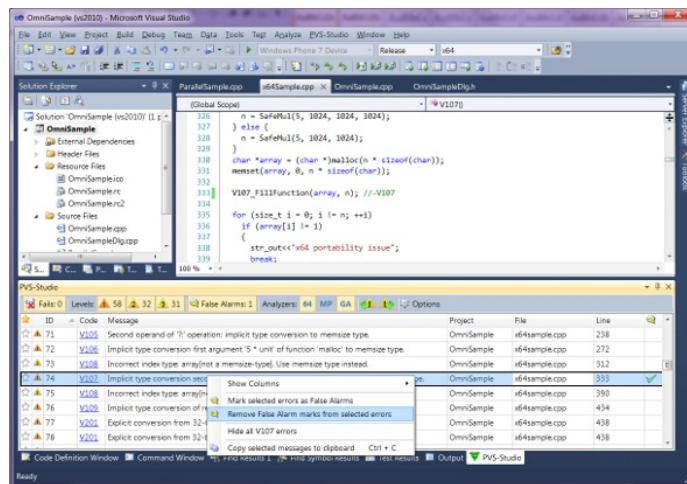


Figure 8 - "Remove False AlarmMarks from selected errors" command

So, we have marked one error as a "false alarm". Let's relaunch the analysis and see that we get one less message. Note that the message we have marked as a "false alarm" is absent in the PVS-Studio window this time.

If you need to see all the messages in the PVS-Studio window (including the false alarms), you may enable their display once again using the "FA" check-button (Figure 9).

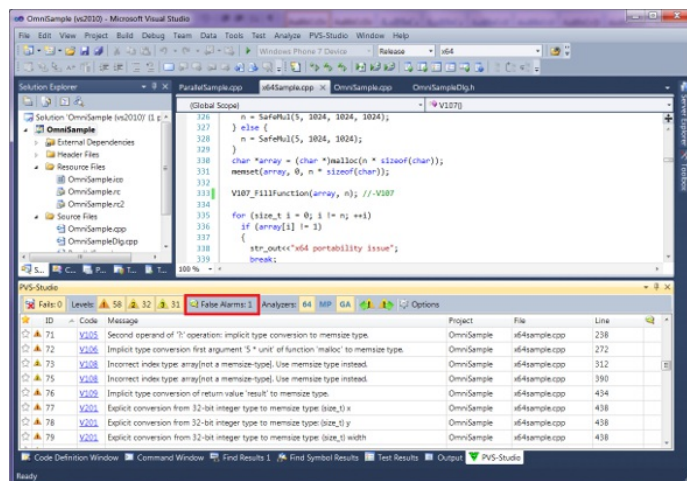


Figure 9 – Enabling the display of marked messages

You may mark several messages at once. To do this, you should choose them in the PVS-Studio window (Figure 10).

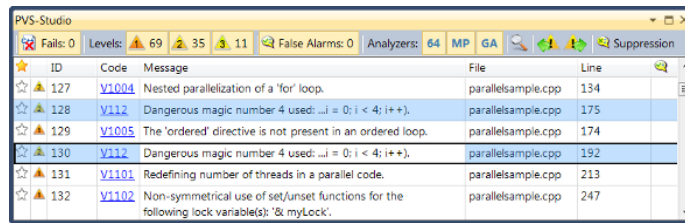


Figure 10 - Choosing several messages to mark in the PVS-Studio window

We do not recommend you to mark messages as false alarms without preliminarily reviewing the corresponding code fragments since it contradicts the ideology of static analysis. Only the programmer can determine if a particular error message is false or not.

Implementation of the false alarm suppression function

Usually compilers employ `#pragma`-directives to suppress individual error messages. Consider a code sample:

```
unsigned arraySize = n * sizeof(float);
```

The compiler generates the following message:

```
warning C4267: 'initializing' : conversion from 'size_t' to 'unsigned
int', possible loss of data x64Sample.cpp 151
```

This message can be suppressed with the following construct:

```
#pragma warning (disable:4267)
```

To be more exact, it is better to arrange the code in the following way to suppress this particular message:

```
#pragma warning(push)
#pragma warning (disable:4267)
    unsigned arraySize = n * sizeof(float);
#pragma warning(pop)
```

The PVS-Studio analyzer uses comments of a special kind. Suppression of the PVS-Studio's message for the same code line will look in the following way:

```
unsigned arraySize = n * sizeof(INT_PTR); //-V103
```

This approach was chosen to make the target code cleaner. The point is that PVS-Studio can inform about issues in the middle of multi-line expressions as, for instance, in this sample:

```
size_t n = 100;
for (unsigned i = 0;
    i < n; // the analyzer will inform of the issue here
    i++)
{
    // ...
}
```

To suppress this message using the comment, you just need to write:

```
size_t n = 100;
for (unsigned i = 0;
    i < n; //-V104
    i++)
{
    // ...
}
```

But if we had to add a `#pragma`-directive into this expression, the code would look much less clear.

Storage of the marking in source code lets you modify it without the risk to lose information about lines with errors.

Theoretically we may also use a separate base where we could store information in the following approximate pattern: error code, file name, line number. The disadvantage of this approach is that you risk losing information about numbers of lines with errors when modifying source files from outside. Storage of the marking in source code allows you to bring modifications into it without the risk to lose information about lines with errors.

Suppressing false positives located within macro statements (`#define`)

It goes without saying that the analyzer can locate potential problems within macro statements (`#define`) and produce diagnostic messages accordingly. But at the same time these messages will be produced by analyzer at such positions where the macro is being used, i.e. where placement of macro's body into the code is actually happening. An example:

```
#define TEST_MACRO \
    int a = 0;      \
    size_t b = 0;   \
    b = a;

void func1()
{
    TEST_MACRO // V101 here
}

void func2()
{
    TEST_MACRO // V101 here
}
```

To suppress these messages you can use the "Mark as False Alarm" command. Then the code containing suppression commands will look like this:

```
#define TEST_MACRO \
    int a = 0;      \
    size_t b = 0;   \
    b = a;

void func1()
{
    TEST_MACRO //-V101
}
```

```
void func2()
{
    TEST_MACRO //-V101
}
```

But in case the macro is being utilized quite frequently, marking it everywhere as False Alarm is quite inconvenient. It is possible to add a special marking to the code manually to make the analyzer mark the diagnostics inside this macro as False Alarms automatically. With this marking the code will look like this:

```
//-V:TEST_MACRO:101

#define TEST_MACRO \
    int a = 0;      \
    size_t b = 0;   \
    b = a;

void func1()
{
    TEST_MACRO
}

void func2()
{
    TEST_MACRO
}
```

During the verification of such a code the messages concerning issues within macro will be immediately marked as False Alarms. Also, it is possible to select several diagnostics at once, separating them by comma:

```
//-V:TEST_MACRO:101, 105, 201
```

Please note that if the macro contains another nested macro inside it then the name of top level macro should be specified for automated marking.

```
#define NO_ERROR 0
#define VB_NODATA ((long)(77))
size_t stat;

#define CHECK_ERROR_STAT \
    if( stat != NO_ERROR && stat != VB_NODATA ) \
        return stat;

size_t testFunc()
{
    {
        CHECK_ERROR_STAT // #1
    }

    {
        CHECK_ERROR_STAT // #2
    }

    return VB_NODATA; // #3
}
```

In the example mentioned above the V126 diagnostics appears at three positions. To automatically mark it as False Alarm one should add the following code at positions #1 and #2:

```
//-V:CHECK_ERROR_STAT:126
```

To make it work at #3 you should additionally specify this:

```
//-V:VB_NODATA:126
```

Unfortunately to simply specify "to mark V126 inside VB_NODATA macro" and not to specify anything for CHECK_ERROR_STAT macro is impossible because of technical specifics of preprocessing mechanism.

Mass suppression of false positive alarms

Let us assume that the following structure exists:

```
struct EXRGBA
{
    unsigned data;
};
```

Also there are several functions that are utilizing it:

```
void f1(const struct EXRGBA aaa)
{
}

long int f2(int b, const struct EXRGBA aaa)
{
    return int();
}

long int f3(float b, const struct EXRGBA aaa, char c)
{
    return int();
}
```

The analyzer produces three V801: *"Decreased performance. It is better to redefine the N function argument as a reference"* messages concerning these functions. Such a message will be a false one for the source code in question, as the compiler will optimize the code by itself, thus negating the issue. Of course it is possible to mark every single message as a False Alarm using the *"Mark As False Alarm"* option. But there is a better way. Adding this line into the sources will suffice:

```
//-V:EXRGBA:801
```

We advise you to add such a line into .h file near the declaration of the structure, but if this is somehow impossible (for example the structure is located within the system file) you could add this line into the stdafx.h as well.

And then, every one of these V801 messages will be automatically marked as false alarm after re-verification.

Other means of filtering messages in the PVS-Studio analyzer

The analyzer also provides three more methods of error messages filtering.

First, you may disable diagnosis of some errors by their code. You may do this using the "[Settings: Detectable Errors](#)" tab. On the tab of detected errors, you may specify the numbers of errors that must not be shown in the analysis report. Sometimes it is reasonable to remove errors with particular codes from the report. For instance, if you are sure that errors related to explicit type conversion (codes V201, V202, V203) are not relevant for your project, you may hide them.

Second, you may disable analysis of some project's parts (some folders or project files). This is the "[Settings: Don't Check Files](#)" tab. On this tab, you may insert information about libraries whose files' inclusions (through the #include directive) must not be analyzed. This might be needed to reduce the number of unnecessary diagnostic messages. Suppose your project employs the Boost library. Although the analyzer generates diagnostic messages on some code from this library, you are sure that it is rather safe and well written. So, perhaps there is no need to get warnings concerning its code. In this case, you may disable analysis of the library's files by specifying the path to it on the settings page. Besides, you may add file masks to exclude some files from analysis. The analyzer will not check files meeting the mask conditions. For instance, you may use this method to exclude autogenerated files from analysis.

Third, you may suppress separate messages by their text. On the "[Settings: Message Suppression](#)" tab, you may set filtering of errors by their text and not their code. If necessary, you may hide error messages containing particular words or phrases in the report. For instance, if the report contains errors that refer to the names of the functions printf and scanf and you think that there cannot be any errors related to them, you should simply add these two words using the editor of suppressed messages.

Possible issues

There might be some issues when using the "Mark as False Alarm" function. Sometimes the analyzer "misses" the number of a line with an error. For instance, the analyzer says that there is an error in line 57 while line 57 is empty at all. Then it is the code, for instance, one line above (line 56) that causes the error.

The reason is that the code analyzer uses the preprocessor from Visual C++ that experiences problems when dealing with multi-line macros (#define). These problems were eliminated in Visual Studio 2005 Service Pack 1 and later versions.

Another issue of the preprocessor refers to multi-line #pragma-directives of a particular type that also cause confusion with line numbering. Unfortunately, this error has not been fixed in any version of Visual Studio yet.

So, markers arranged automatically might sometimes appear in false places. In this case, the analyzer will again produce the same error warnings because it will fail to find the markers. To solve this issue, you should mark messages you experience troubles with manually. PVS-Studio always informs about such errors with the message "[V002. Some diagnostic messages may contain incorrect line number](#)".

Like in case of any other procedure involving mass processing of files, you must remember about possible access conflicts when marking messages as false alarms. Since some files might be opened in an external editor and modified there during file marking, the result of joint processing of such files cannot be predicted. That is why we recommend you either to have copies of source code or use version control systems.

The principle of constantly using PVS-Studio in the process of project development

Introduction of "Mark as False Alarm" function into PVS-Studio significantly enhances the potential of code analyzer's integration into the software development process. Now the analyzer can be conveniently launched not from time to time but every day, for instance.

The mechanism of using the analyzer in this way consists of two steps: introduction and constant use. Suppose we have a large project of several millions of source code lines. There are twenty programmers participating in the project all the time. Suppose we need to port this project to a 64-bit system (The procedure of using PVS-Studio parallel code analyzer is absolutely the same, but to make it clearer, let us talk about developing a 64-bit version).

We advise you to use our tool in this way.

At first, a team of five developers creates the base compilable configuration of the 64-bit version of the product, fixes the basic errors and compiler's warnings. After that, the project is split into parts shared among the developers and PVS-Studio analyzer is launched. For some time (days/weeks/months), depending on the size of the source code, the team either fixes the problems detected (the analyzer's warnings) or mark them False Alarm in the code. As a result, when the work on code migration is over, PVS-Studio will generate no new warnings because all the errors are either corrected or marked as false alarms.

After that, the other 15 developers begin working on the project (and its further development). While creating a new code, they can launch the analyzer every day and fix possible errors only in the new code. And they will not see the previous, already treated, warnings of the analyzer.

This approach allows you not only to port an application to a 64-bit platform, but also to make sure that there are no 64-bit problems in the freshly developed code.

Naturally, you can use PVS-Studio to search for parallel programming errors in the same way. The first step of analyzing and marking false alarms is performed by a small team of developers. After that, PVS-Studio can be used by the whole team who can consider errors only in the new code they are responsible for.

Handling the diagnostic messages list

[Navigation and sorting](#)

[Searching for individual messages. quick jumps](#)

[Managing the Visual Studio Task List](#)

While handling the large number of messages (and the first-time verification of large-scale projects, when filters have not been set yet and false positives haven't been marked, the number of generated messages can come close to tens of thousands), it is reasonable to use the navigational and searching mechanisms integrated into PVS-Studio output window.

Navigation and sorting

The main purpose of PVS-Studio output window is to simplify the analyzed project's source code navigation and reviewing of potentially dangerous fragments in it. Double-clicking any of the messages in the list will automatically open the file corresponding to this message in the code editor and will place the cursor on the desired line. The quick navigation buttons (see figure 1) allow for an easy review of the potentially dangerous fragments in the source code without the need of constant IDE windows switching.



Figure 1 — Quick navigation buttons

To present the analysis results, PVS-Studio output window utilizes a virtual grid, which is capable of fast rendering and sorting of generated messages even for huge large-scale projects (virtual grid allows you to handle a list containing hundreds of thousands of messages without any considerable hits to performance). The far left grid column can be used to mark messages you deem interesting, for instance the ones you wish to review later. This column allows sorting as well, so it won't be a problem to locate all the messages marked this way. The "Show columns" context menu item can be used to configure the column display in the grid (figure 2):

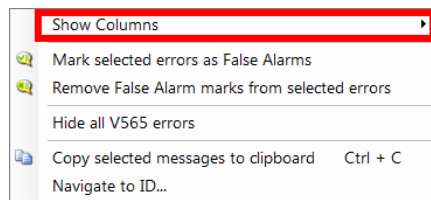


Figure 2 — Configuring the output window grid

The grid supports multiline selection with standard Ctrl and Shift hotkeys, while the line selection persists even after the grid is resorted on any column. The "Copy selected messages to clipboard" context menu item (or Ctrl+C hotkey) allows you to copy the contents of all selected lines to a system clipboard.

Searching for individual messages, quick jumps

PVS-Studio output window contains the integrated mechanisms for quick keyword searching and navigating to individual messages in the grid. The search window can be opened by "Find in PVS-Studio Output" button (Figure 3).

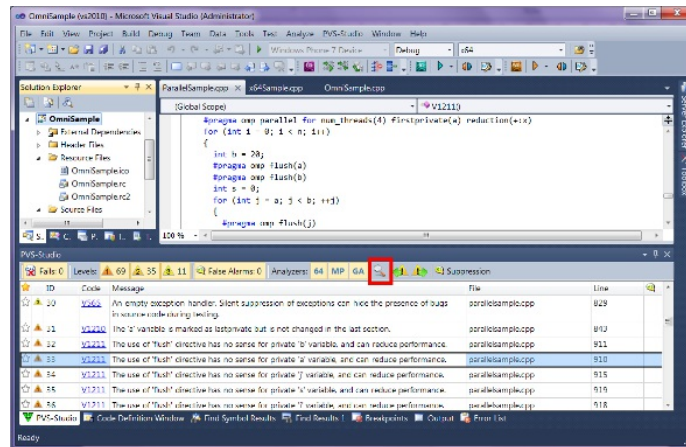


Figure 3 — Opening the search window

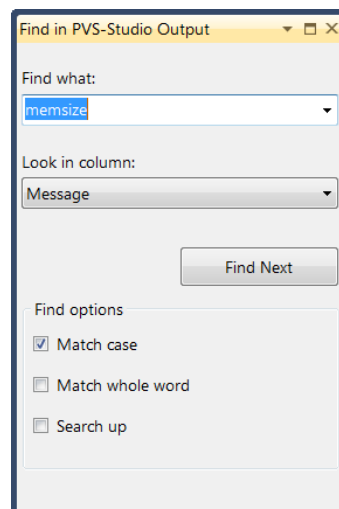


Figure 4 — PVS-Studio search window

Clicking the "Find Next" button selects and auto-focuses the next (or previous) line relative to the current selection, in which the specified keyword was discovered. It's worth noting that keyword searching is performed only for the column selected in the drop-down list of the "look in column" combobox.

In case there is a need to navigate to an individual message in the grid, it is possible to use the quick jumping dialog, which can be accessed through the "Navigate to ID..." context menu item (figure 5):

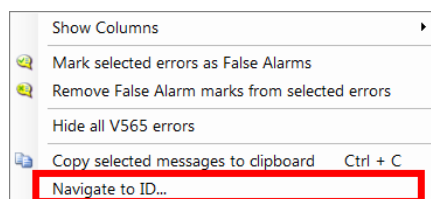


Figure 5 — evoking the quickjumping dialog

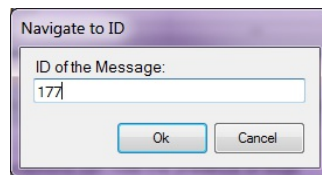


Figure 6 – Navigate to ID dialog

Each of the messages in PVS-Studio output list possesses a unique identifier — the serial number under which this message was added into the grid, which itself is displayed in the ID column. The quick navigation dialog allows you to select and auto-focus the message with the designated ID, regardless of current grid's selection and sorting. You also may note that the IDs of the messages contained within the grid are not necessarily strictly sequential, as a fraction them could be hidden by the filtering mechanism, so navigation to such messages is impossible.

Managing the Visual Studio Task List

The large-scale projects are often developed by a distributed team, so a single person isn't able to judge every message static analyzer generates for false-positives, and even more so, is unable to correct the corresponding sections of the source code. In this case it makes sense to delegate such messages to a developer who is directly responsible for the code fragment in question.

PVS-Studio allows you to automatically generate the special TODO comment containing all the information required to analyze the code fragment marked by it, and to insert it into the source code. Such comment will immediately appear inside the Visual Studio Task List window (in Visual Studio 2010 the comments' parsing should be enabled in the settings: Tools->Options->Text Editor->C++->Formatting->Enumerate Comment Tasks->true) on condition that the 'Tools->Options->Environment->Task List->Tokens' list does contain the corresponding TODO token (it is present there by default). The comment could be inserted using the 'Add TODO comment for Task List' command of the context menu (figure 7):

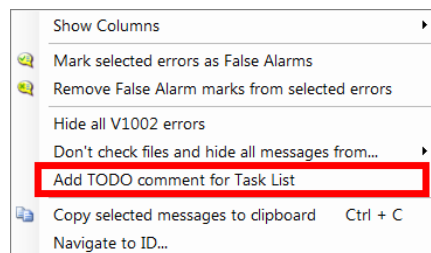


Figure 7 – inserting the TODO comment

The TODO comment will be inserted into the line which is responsible for generation of analyzer's message and will contain the error's code, analyzer message itself and a link to the online documentation for this type of error. Such a comment could be easily located by anyone possessing an access to the sources thanks to the Visual Studio Task List. And with the help of the comment's text itself the potential issue could be detected and corrected even by the developer who does not have PVS-Studio installed or does not possess the analyzer's report for the full project (figure 8).

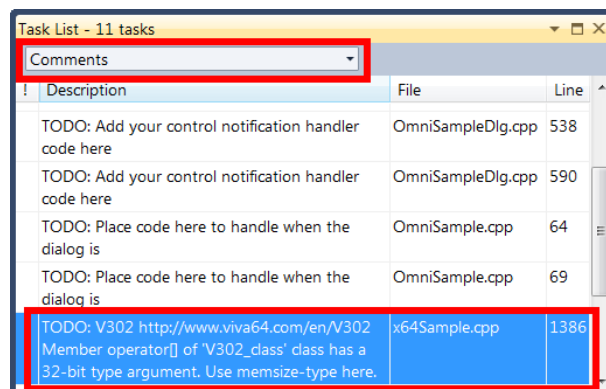


Figure 8 —Visual Studio Task List

The Task List Window could be accessed through the View->Other Windows->Task List menu. The TODO comments are displayed in the 'Comments' section of the window.

OmniSample, a collection of examples of 64-bit and parallel errors

[Abstract](#)

[Introduction](#)

[How to get OmniSample](#)

[The project's structure](#)

[Using the program](#)

[Introduction into the project's code](#)

Abstract

The article presents a review of the OmniSample application which is a collection of examples of 64-bit and parallel errors diagnosed by the PVS-Studio static code analyzer. OmniSample is shipped together with the PVS-Studio distribution package.

Introduction

The PVS-Studio tool is a set of static analyzers of C++ source code. It includes the Viva64 unit intended for detecting 64-bit migration errors, VivaMP unit that

diagnoses errors in concurrent programs written with the OpenMP technology and general purpose unit. Despite the fact that various reference documentation for programmers provides description of these errors, it often comes just to generalities. At the same time, there are rather few articles containing examples of errors in application code. The OmniSample project is intended to make up this deficiency and also show all the diagnostic capabilities of the PVS-Studio tool.

How to get OmniSample

The OmniSample project is shipped together with the PVS-Studio analyzer. To get the project you need just to download the distribution package from [our site](#) and install it. After that you will get a link to the zip-archive (64-bit and Parallel issues example) in the "PVS-Studio" section of the "Start" menu containing OmniSample.

The project's structure

OmniSample is a set of C++ projects for Visual Studio 2005, Visual Studio 2008 and Visual Studio 2010 development environments. The OmniSample project includes samples of all the errors diagnosed by the Viva64 and VivaMP analyzers. Each of these samples is represented by a separate function and all these functions are in their turn grouped in files: x64Sample.cpp (for 64-bit errors) and ParallelSample.cpp (for parallel errors). Each example of errors diagnosed by the analyzers corresponds to 2 functions: the one with the error and the correct one. Note that the correct version of the function here contains the fixed version of the incorrect function's code.

For optimal work of the program with 64-bit examples, it is better to have not less than 6 Gbytes of main memory. If you launch these examples in a system with less memory, you might encounter long "hangs" of the operating system related to swap file operations.

Using the program

It is convenient to use the PVS-Studio analyzer to study the samples presented in the program. Analysis results for one of the OmniSample projects, while being opened in the corresponding version of Visual Studio IDE, will let you quickly find the error you want to study and jump to the code line containing it.

Note one important thing about using the demo version of PVS-Studio. When you use the demo version to check code, PVS-Studio finds all the potential errors but does not show all information about their location in the program source code in its diagnostic messages. In such cases, you will see the text "TRIAL RESTRICTION" instead of the line's number as shown in Figure 1.

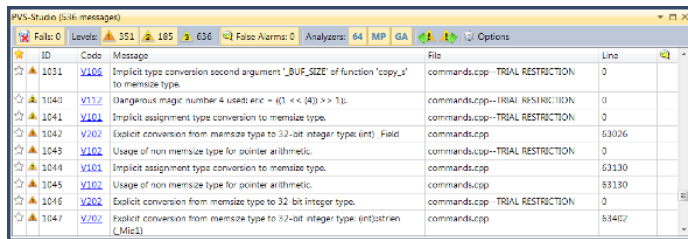


Figure 1 – Using the demo version of PVS-Studio when checking a project does not allow you to see positions of all the errors in the code

When the demo version of PVS-Studio deals with the OmniSample project, it shows all the errors. That is, PVS-Studio does not have any restrictions when working with samples. You can freely modify OmniSample files containing errors, so that you could fully study PVS-Studio's behavior on the code you have written by yourself.

To get a detailed description of each analyzer-generated warning and techniques required to correct the corresponding error, you may select the message you need and click on its error code or read this description in the [PVS-Studio documentation](#) section on our site.

The interface of the OmniSample program (Figure 2) makes it convenient for you to launch only those functions that contain code fragments you need.

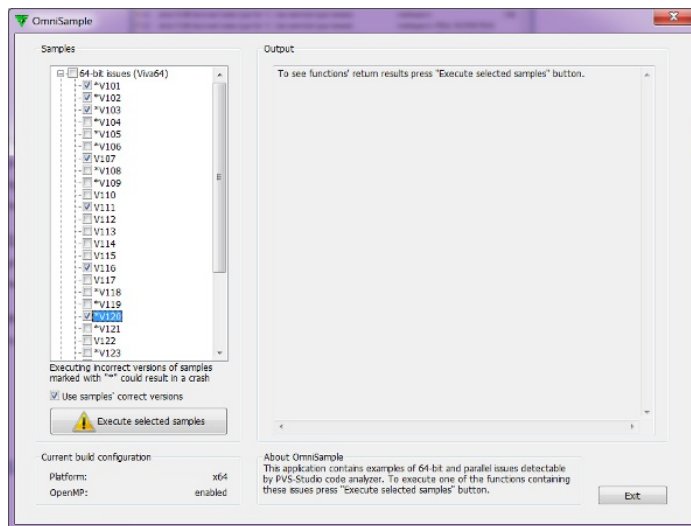


Figure 2 – The interface of the OmniSample program

To do this, select the numbers of PVS-Studio messages you need in the tree-like list on the left and click the "Execute selected samples" button. Note that all the functions are grouped according to error types, i.e. 64-bit and parallel errors. To execute all the samples of errors referring to one type, you need just to select the corresponding branch of the list. Results of selected functions' execution are displayed in the "Output" field to the right of the error list (Figure 3).

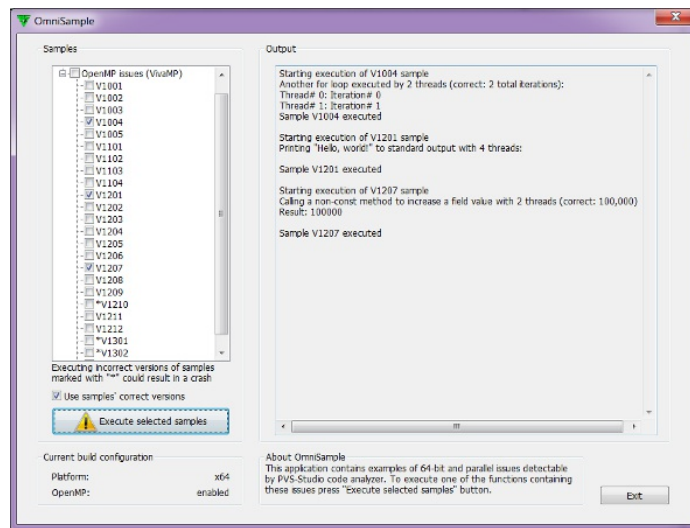


Figure 3 – Execution results of V1004, V1201 and V1207 functions

By default, the program will execute the code of correct (fixed) versions of the selected functions. To execute non-corrected samples, you should uncheck the "Use sample's correct versions" option below the list. Remember that such functions might cause a program crash. Samples whose incorrect versions cause OmniSample to crash are marked with the "*" symbol.

The "Current build configuration" field represents current parameters of the OmniSample project's build. To study samples of parallel errors, you should build the project using the OpenMP standard (the /openmp compiler switch). To study samples of 64-bit errors, you should build the project in the 64-bit configuration (x64).

Introduction into the project's code

Let's review the code of the functions that correspond to the V102 message to show how to work with the OmniSample project. At first let's study the `std::string V102()` function that contains a 64-bit error corresponding to the V102 warning:

```
std::string V102()
{
    std::ostringstream str;

    int domainWidth;
    int domainHeght;
    int domainDepth;
    if (IsX64Platform()) {
        domainWidth = 1000;
        domainHeght = 1000;
        domainDepth = 3000;
    } else {
        domainWidth = 100;
        domainHeght = 100;
        domainDepth = 300;
    }

    char *buffer =
        new char [size_t(domainWidth) * size_t(domainHeght) * size_t(domainDepth)];

    char *current = buffer;
    char *end = buffer;
    end += domainWidth * domainHeght * domainDepth;

    while (current != end)
        *current++ = 1;

    delete [] buffer;

    return str.str();
}
```

After the project is checked by the Viva64 analyzer, it will detect the error "V102 Usage of non memsize type for pointer arithmetic" in the following line:

```
end += domainWidth * domainHeght * domainDepth;
```

The issue of this code lies in pointer arithmetic, or rather in using non-memsize types for this arithmetic. The error here is that the "end" pointer will never get an increment of more than 4 Gbytes on the 64-bit platform. In the correct version of this function, `std::string V102_correct()`, it is the memsize-type `ptrdiff_t` which is used for the variables `domainWidth`, `domainHeght` and `domainDepth`:

```
std::string V102_correct()
{
    std::ostringstream str;

    ptrdiff_t domainWidth;
    ptrdiff_t domainHeght;
    ptrdiff_t domainDepth;
    if (IsX64Platform()) {
        domainWidth = 1000;
        domainHeght = 1000;
        domainDepth = 3000;
    } else {
        domainWidth = 100;
        domainHeght = 100;
        domainDepth = 300;
    }

    char *buffer =
        new char [size_t(domainWidth) * size_t(domainHeght) * size_t(domainDepth)];

    char *current = buffer;
    char *end = buffer;
    end += domainWidth * domainHeght * domainDepth;

    while (current != end)
```

```

    *current++ = 1;

delete [] buffer;

return str.str();
}

```

You may view the whole source code of both 64-bit and parallel errors including correct versions of code.

Checking code with PVS-Studio from the command line (with a Visual Studio solution file present)

[Abstract](#)

[Introduction](#)

[Common launch of the tool from the command line with the Visual Studio window](#)

[Launching the analyzer in batch mode through command line](#)

[Impact of PVS-Studio settings file on the procedure of command line launch](#)

[Regular use of PVS-Studio and integration into the process of "everyday builds"](#)

[Launching PVS-Studio from the command line without using Visual Studio](#)

[Summary](#)

Abstract

In this article we will show you how to check solutions with PVS-Studio from the command line instead of Visual Studio environment. It is a new capability of PVS-Studio 3.50. Note that we still mean that the checking will be performed from Visual Studio involving the files of projects (.vcproj) and solutions (.sln) but it will be launched from the command line instead of IDE. This way of launching the tool may be useful when you need to regularly check the code with the help of build systems or continuous integration systems.

Introduction

[PVS-Studio](#) is a static code analyzer intended for developers of modern 64-bit and parallel C/C++ applications. Development of such programs implies some difficulties different from the issues of the traditional programming, because there are specific types of problems besides common errors (like uninitialized pointers) which are known to every programmer and may be detected by a compiler.

We will not speak too much about the tool PVS-Studio itself and its purpose in this paper. You may find this information in the review article "[Getting acquainted with PVS-Studio code analyzer](#)" and a more solid "[PVS-Studio Tutorial](#)". Here we want to focus on the task of using the tool from the command line but involving Visual Studio. This way of using the tool is employed by developer teams that use either build systems (it does not matter what they are based on - like Make, NMake, NAnt, MSBuild) or continuous integration systems (like Cruise Control, Draco.NET or Team Foundation Build from Visual Studio Team System).

Common launch of the tool from the command line with the Visual Studio window

To become acquainted with the process of working with PVS-Studio, we advise you to install the project *OmniSample* from the PVS-Studio distribution kit and test the analyzer with it. Let us assume that the project *OmniSample* is installed into the folder C:\Users\evg\Documents\OmniSample. We need to test this project in Visual Studio 2008. So we just need to write in the command line (the arguments containing directory paths should not end with the \ symbol, as the combination \\" will not always be correctly processed by the OS):

```

"C:\Program Files (x86)\PVS-Studio\x64\PVS-Studio.exe" --sln-file "C:\Users\evg\Documents\ OmniSample\OmniSample (vs2008).sln" --plog-file
"C:\Users\evg\Documents\result.plog" --vcinstalldir "C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC" --platform "x64" --configuration "Release"

```

The *devenv.exe* file of the required version will be launched, then the solution "OmniSample (vs2008).sln" will be opened, the x64 architecture will be selected as well as the Release configuration, and the analysis of the solution in Visual Studio will start. When PVS-Studio's work is done, the report will be saved in the file C:\Users\evg\Documents\result-log.plog and Visual Studio will be closed. Then you may open the saved report in Visual Studio clicking the command "Load Analysis Report" in PVS-Studio menu or by double-clicking it directly from file manager and review it through in the convenient environment with the function of code navigation available, filters and all the PVS-Studio functions enabled. In other words you could and should work with the report file in the PVS-Studio because it is created in a special format.

If the project is correct and everything is set right, the analysis will be over in some time, the window will be closed and you may open the file *result-log.plog*. But if something is wrong, you will see a window with a warning message. Unfortunately, Visual Studio cannot pass the messages to the console, so the analysis may sometimes get 'hanged' because of the messages generated during night builds when the tool waits for the programmer to respond. We can hardly fix this issue somehow. The only thing we may recommend you is to run the analysis with the user present at first.

If usage of the long command line argument is inconvenient for some reasons, it is possible to set all these parameters through special PVS-Studio configuration file and pass a path to this file as an argument:

```
PVS-Studio.exe --cfg "PVS-Studio.cfg"
```

PVS-Studio.cfg file contains:

```
sln-file = C:\Users\evg\Documents\OmniSample\OmniSample (vs2008).sln
```

```
plog-file = C:\Users\evg\Documents\OmniSample\result.plog
```

```
vcinstalldir = C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC
```

```
platform = x64
```

```
configuration = Release
```

Launching the analyzer in batch mode through command line

Launching through command line will by default make the analyzer check all the files included in selected Visual Studio solution. If, however, the verification of only a predefined set of files is required, it is possible to utilize the batch mode which will subject to analysis only files that were explicitly defined by their paths. To start the analysis in batch mode you should specify the path to a list of files required for verification as the 4th command line argument (this argument is optional).

To verify files in batch mode from command line you should start the analysis using these arguments:

```

"C:\Program Files (x86)\PVS-Studio\x64\PVS-Studio.exe" --sln-file "C:\Users\evg\Documents\ OmniSample\OmniSample (vs2008).sln" --plog-file
"C:\Users\evg\Documents\result.plog" --vcinstalldir "C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC" --platform "x64" --configuration "Release" --filelist-file

```


"C:\Users\evg\Documents\OmniSample\ExtFileList.txt"

The presence of all C++ source files inside projects of selected sln file is a prerequisite.

The list from *filelist-file* should contain full paths to C++ files separated by newline (\r\n) characters. The allowed file resolutions are c, cc, cpp and cxx, i.e. the files which can be compiled.

Below is the example of ExtFileList.txt text file (each file located on separate line):

c:/Users/evg/Documents/OmniSample/OmniSample/./OmniSample/OmniSample.cpp

"c:\Users\evg\Documents\OmniSample\OmniSample\.\OmniSample\ParallelSample.cpp"

Impact of PVS-Studio settings file on the procedure of command line launch

When you launch code analysis from the command line, settings are used are all the same as in the case of launching from Visual Studio itself - actually, it is a launch from Visual Studio. And it is the same number of the processor cores being used that is specified in the settings.

Concerning the system of filters (Message Suppression and Detectable Error), it is NOT used when you launch analysis from the command line. It means that the report file will contain all the warnings about the errors regardless of the filtration settings. But when you load the report file into Visual Studio, the filters will be enabled. This behavior is explained by the fact that the filters are applied to the results dynamically (when launching from Visual Studio as well). It is very convenient since you may want to disable some of the warnings (V201, for instance) after getting the warning list. You just need to disable them in the settings and the corresponding warnings will disappear from the list WITHOUT requiring you to relaunch the analysis.

Regular use of PVS-Studio and integration into the process of "everyday builds"

If you regularly use PVS-Studio, you should exploit the function "Mark as False Alarm". This function lets you mark those diagnostic warnings of PVS-Studio analyzer that are "false alarms" to avoid showing them at the next launch of the analyzer. For example, the analyzer generated the warning V101 on the following code fragment but ceased to do that after adding a special comment:

```
size_t bufferSize = imageWidth * imageHeight * //-V101
                  bytePerPixel * maxFrameCountInBuffer;
```

You may add this comment manually but we advise you to use a special command of PVS-Studio menu (or toolbar).

The process of working with PVS-Studio code analyzer consists of two stages: integration and constant use. At the stage of integrating PVS-Studio into an existing large project programmers look through the analyzer-generated warnings and either correct the code or mark it with the help of the function "Mark as False Alarm". On making it all clear, the developers launch the final analysis of the whole code and get 0 warnings from the analyzer (unless the display of errors marked as false alarms is enabled). It means that the integration stage is complete and the stage of constant use begins.

From this moment on, all new code that will be added into the project will be tested by PVS-Studio. Actually, PVS-Studio will test the WHOLE code but will detect only new errors - the ones in a newly written or corrected code or obsolete code that has not been marked with the help of the function "Mark as False Alarm".

The capability of launching code analysis from the command line is needed to perform regular (for example, everyday) check of the code. The procedure looks in this way:

You launch the code analysis from the command line.

Then you send the final report file to all the developers concerned.

When you launch PVS-Studio from the command line, besides the xml-file (for instance, result-log.plog) there is one more text file created whose name ends with ".only_new_messages.txt" (for example, result-log.plog.only_new_messages.txt). This file contains new (not marked as false alarms) messages from PVS-Studio:

c:\users\evg\documents\omnisample\x64Sample.cpp(17): error V101: Implicit assignment type conversion to memsize type.

c:\users\evg\documents\omnisample\x64Sample.cpp(50): error V201: Explicit type conversion. Type casting to memsize.

c:\users\evg\documents\omnisample\x64Sample.cpp(54): error V102: Usage of non memsize type for pointer arithmetic.

If there are no new messages, the file result-log.plog.only_new_messages.txt will contain the phrase:

No new diagnostic messages were generated.

The xml-file will contain the same number of messages in the both cases while the txt-file only those that have not been marked as "False Alarm".

When launching PVS-Studio every day, the existing txt-file with new warnings from PVS-Studio will be small and you may send it to all the developers concerned via e-mail. But it is not reasonable to send the xml-file via e-mail because it contains full paths to the files. But if the developers have the same paths to the files on their computers, you may send xml-files as well.

Thus, you may avoid new errors in your code launching PVS-Studio regularly every day.

Launching PVS-Studio from the command line without using Visual Studio

The way of launching PVS-Studio from the command line we have discussed in the article can be used ONLY if you have project (.vcproj) and solution (.sln) files which are correctly adjusted. If you do not have these files but still want to test the code, see the article "[Using PVS-Studio from the command line](#)".

Summary

Everyday launch of PVS-Studio from the command line may help you significantly increase the code quality, especially if the code is marked using the function "Mark As False Alarm". In this case you will see 0 new warnings every day if there are no errors. If the already checked code is incorrectly modified, you will see the warnings about these flaws at the next launch of PVS-Studio. Of course, the new code will also be regularly tested in the automatic mode (independently from the programmer). It will allow you to create quality 64-bit and parallel applications.

Using PVS-Studio from the command line (without Visual Studio solution file)

[Abstract](#)

[Introduction](#)

[What should you know before launching the tool from the command line?](#)

[Specifics of using PVS-Studio while launching from command line](#)

[PVS-Studio analyzer independent mode](#)

[Using Microsoft IntelliSense with analyzer in independent mode.](#)

[An example of using the analyzer independent mode with Microsoft NMake project](#)

[Conclusion](#)

Abstract

PVS-Studio code analyzer used to develop modern 64-bit and parallel applications perfectly integrates into Microsoft Visual Studio environment. But sometimes, there are situations when using the tool in the environment is impossible or inconvenient. The article provides guidelines and examples from practice on using the analyzer in the command line-launch mode. The article describes the 4.30 version of PVS-Studio analyzer.

Introduction

[PVS-Studio](#) is a static code analyzer intended for developers of modern resource-intensive applications in C and C++ languages. By "modern applications" we understand 64-bit and/or parallel applications. Development of these programs implies some difficulties different from those of traditional programming. Because besides common errors known to everyone like uninitialized pointers, detected by the compiler, there are new types of issues.

In this article, we won't dwell upon description of PVS-Studio tool and its purposes. To learn this, refer to the review article "[Getting acquainted with PVS-Studio code analyzer](#)" and a more solid "[PVS-Studio Tutorial](#)". This article's purpose is to discuss the use of the tool from the command-line.

We recommend that everyone should use PVS-Studio in Microsoft Visual Studio development environment into which the tool is perfectly integrated. But sometimes you can face situations when you should launch it using the command line. In case you possess project (.vcproj) and solution (.sln) files, and command line execution is required for the sake of daily code checks, for instance, we advise you to examine the article "[Checking code with PVS-Studio from the command line \(with a Visual Studio solution file present\)](#)". Otherwise you should consider using this article.

What should you know before launching the tool from the command line?

To be able to launch the tool from the command line successfully it is extremely important to understand the code analyzer's working principles. When working in Visual Studio, much of this stays hidden (and it is better for the user this way). But when launching the tool from the command line one must understand all the details and peculiarities.

So, how does a code analyzer work (be it PVS-Studio or any other tool)?

When the analyzer user gives a command to check some file (for example, file.cpp), the analyzer performs [preprocessing](#) of this file at first. As a result, all the macros are defined and #include-files are arranged. Preprocessing during code analysis is necessary for the analyzer to be able to "see" definitions of functions, types and classes which will be used in the analyzed file then. Some code analyzers have an internal preprocessor. Others (like PVS-Studio) use an external one. We use the preprocessor from Visual Studio compiler. The result of preprocessing is the file with the extension ".i". As a rule, this file is rather large as it contains the bodies of all the header files used in the module.

The preprocessed i-file can now be parsed by the code analyzer. Pay attention that the analyzer cannot parse a file which has not been preprocessed, for it won't have information about the types, functions and classes being used.

So, operation of any code analyzer includes at least two steps: preprocessing and analysis itself.

Specifics of using PVS-Studio while launching from command line

PVS-Studio tool has been developed to work within the framework of Visual Studio environment. And launching it from the command line is the function that is additional to the main working mode. However, all of analyzer's diagnostic capabilities are available. The only significant difference between operating within Visual Studio and launching from the command line is the following.

Suppose we have a project consisting of three files: file.h, file1.cpp and file2.cpp. And file.h is included both in file1.cpp and file2.cpp through #include directive. Suppose PVS-Studio analyzer found an error inside file.h when analyzing file1.cpp and informed about it. When analyzing file2.cpp, the same error will be found inside file.h (for file.h is included into the both files). But when launching the tool in Visual Studio, the error won't be shown for the second time as there is a mechanism filtering repeating warnings. When launching the analyzer from the command line, this mechanism doesn't work and the warning will be shown in the both cases - when analyzing file1.cpp and file2.cpp.

Also, when launching from the command line, the mechanisms of error-warning filtration are not available (see: [diagnosable errors](#) and [suppression of separate warnings](#)).

However the error messages generated in this mode could be easily redirected into the external file with the help of **--output-file** command line switch. This file will contain the unprocessed and unfiltered analyzer output.

Such a file could be viewed in PVS-Studio toolwindow of Visual Studio IDE using 'Load Analysis Report' menu command (selecting 'Unparsed output' as a file type) and afterwards it could be saved in a standard PVS-Studio log file (plog) format. This allows you to avoid the duplication of error messages and also to use all of the standard filtering mechanisms for them.

Please note, that starting from PVS-Studio version 4.52, the analyzer supports multi-process (PVS-Studio.exe) output into a single file (specified through **--output-file**) in command line independent mode. This allows several analyzer processes to be launched simultaneously during the compilation performed by a makefile based system. The output file will not be rewritten and lost, as file blocking mechanism had been utilized.

PVS-Studio analyzer independent mode

Usually the programs' sources developed with Visual C++ do possess project files (files with vcproj/vcxproj extensions) which in turn contain metadata about their respective structures and build configurations as well as their building parameters. Such sources can be easily analyzed with PVS-Studio using Visual Studio extension plug-in. But it is possible that sources written for Visual C++ compiler do not have project files associated with them, for example it is possible in case of multiplatform software or old projects which are built using command line batch utilities. Various Make systems are often employed to control building process in such cases, Microsoft NMake or GNUMake for instance.

To analyze such projects it is necessary to embed the direct call for the analyzer (the 'programfiles%\PVS-Studio\x64\PVS-Studio.exe' file) into building process and to pass all arguments required for preprocessing to it. The PVS-Studio analyzer should be called in batch mode for each C/C++ file (files with c/cpp/cxx etc. extensions, the analyzer shouldn't be called for header h files) with the following arguments:

```
PVS-Studio.exe --cl-params %ClArgs%  
--source-file %cppFile% --cfg %cfgPath% --output-file %ExtFilePath%
```

%ClArgs% — arguments which are passed to cl.exe compiler during regular compilation.

%cppFile% — path to analyzed C/C++ file

%ClArgs% and %cppFile% parameters should be passed to PVS-Studio analyzer in the same way in which they are passed to Visual C++ compiler.

%cfgPath% — path to PVS-Studio configuration file. This file is shared between all C/C++ files and can be created manually (the example will be presented

below)

%ExtFilePath% — optional argument, a path to the external file in which the results of analyzer's work will be stored. In case this argument is missing, the analyzer will output the error messages into stdout. The results generated here can be viewed in Visual Studio's 'PVS-Studio' toolwindow using 'PVS-Studio/Load Analysis Report' menu command (selecting 'Unparsed output' as a file type).

In fact the analyzer should be called for the same files for which the compiler (cl.exe in case of Visual C++) is being called. It is important to remember that PVS-Studio analyzer works only with Microsoft Visual C++ preprocessed C/C++ files at this moment and therefore can be embedded into building process only if the support for this compiler is present.

The analysis results for each file will be egested into stdout/stderr.

Using Microsoft IntelliSense with analyzer in independent mode.

Although it is possible to open the unfiltered text file containing analyzer diagnostic messages from within the IDE into PVS-Studio Output window (which itself will allow you to use file navigation and filtering mechanisms), you will only be able to use the code text editor inside the Visual Studio itself, as the additional IntelliSense functionality will be unavailable (that is, autocompletion, type declarations and function navigation, etc.). And all this is quite inconvenient, especially while you are handling the analysis results, even more so with the large projects, forcing you to search class and method declarations manually. As a result the time for handling a single diagnostic message will be greatly increased.

To solve this issue, you need to create an empty Visual C++ project (Makefile based one for instance) in the same directory with C++ files being verified by the analyzer (vcproj/vcxproj file should be created in the root folder which is above every file verified). After creating an empty project you should enable the 'Show All Files' mode for it (the button is in the upper part of the Solution Explorer window), which will display all the underlying files in the Solution Explorer tree view. Then you will be able to use the 'Include in Project' context menu command to add all the necessary c, cpp and h files into your project (You will also probably have to add include directory paths for some files, for instance the ones containing third-party library includes). If including only a fraction of the files verified, you also should remember that IntelliSense possibly will not recognize some of the types from within these files, as these types could be defined right in the missing files which were not included by you.

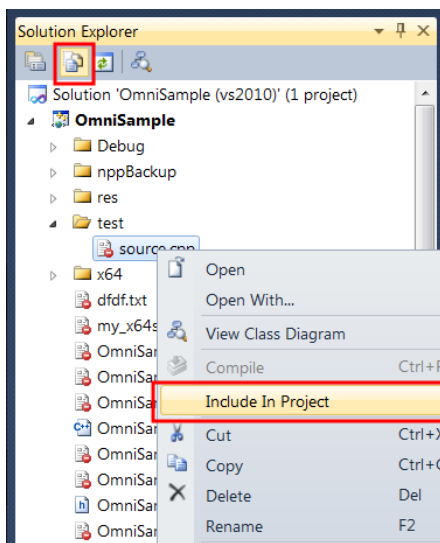


Figure 1 — including files into the project

The project file we created could not be used to build or verify the sources with PVS-Studio, but still it will substantially simplify handling of the analysis results. Such a project could also be saved and then used later with the next iteration of analyzer diagnostics results in independent mode.

An example of using the analyzer independent mode with Microsoft NMake project

For example let's take Microsoft Nmake project which is build using VisualC++ compiler and it is declared in the project's makefile like this:

\$(CC) \$(CFLAGS) \$**

The **\$(CC)** macro calls cl.exe, the compilation parameters **\$(CFLAGS)** are passed to it and finally all C/C++ files on which the current build target is dependent are called in batch mode using **\$**** macro. Thereby the cl.exe compiler will be called with required compilation parameters for all source files.

Let's modify this script in such a way that every file is analyzed with PVS-Studio before the compiler is called (in one line):

```
$(PVS) --source-file $** --cl-params $(CFLAGS) $**  
--cfg "C:\CPP\PVS-Studio.cfg" & $(CC) $(CFLAGS) $**
```

\$(PVS) – path to analyzer's executable (%programfiles%\PVS-Studio\x64\PVS-Studio.exe). Take into account that the Visual C++ compiler is being called after the analyzer on the same line with the same arguments as before. This is done to allow for all targets to be built correctly so the build would not stop because of the lack of .obj-files.

The PVS-Studio.cfg configuration file should include the following lines:

```
exclude-path = C:\Program Files (x86)\Microsoft Visual Studio 10.0  
vcinstalldir = C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\  
platform = Win32
```

Let's review these parameters:

- The exclude-path parameter contains the directories for which the analysis will not be performed. If the Visual Studio directory is not included here, the analyzer will generate error messages for its' header .h-files. But you of course cannot modify them. Therefore we recommend you to always add this path to the exclusions. It is also possible to set multiple exclude-path parameters.
- The vcinstalldir parameter indicates the directory in which the Microsoft Visual C++ compiler is located. This is required to be able to locate the cl.exe compiler and use it for preprocessing.
- The platform parameter points us to the correct version of the compiler — Win32, x64, Itanium or ARMV4. It is usually Win32 or x64.

You can filter diagnostics messages generated by analyzer using analyzer-errors and analysis-mode parameters (set them in cfg file of pass through command line). These parameters are optional.

- The analyzers-errors parameter allows you to set the codes for errors in which you are interested. For example: analyzer-errors=V112 V111. We do not recommend setting this parameter.

- The analysis-mode parameter allows you to control the analyzers being used. Values: 0 – full analysis (by default), 1 – only 64 bit analysis, 2 – only OpenMP analysis, 4 – only general-purpose analysis. The recommended value is 4.

The full list of command line switches will be displayed with this argument:

```
PVS-Studio.exe --help
```

Conclusion

Although the abilities included into PVS-Studio are enough to use the tool from the command line, of course it can be greatly improved. We are ready to improve the mechanism of launching PVS-Studio from the command line to the level allowing making it convenient for you to use the tool in your particular project. Write to us.

Using PVS-Studio with continuous integration systems

[Abstract](#)

[Introduction](#)

[Using the "Mark As False Alarm" function during testing automation](#)

[Integrating PVS-Studio into build automation process](#)

[Launching the analyzer in batch mode through command line](#)

[Integrating with CruiseControl .NET](#)

[Integrating with Hudson](#)

[Integrating with Microsoft Team Foundation Server 2005/2008/2010](#)

[Team Foundation 2010 Build Server](#)

[Team Foundation 2005/2008 Build Server](#)

[Starting Team Foundation builds in interactive desktop mode.](#)

[The analysis of projects containing Team Foundation Server source control.](#)

Abstract

This article illustrates techniques required to employ the use of PVS-Studio static code analyzer together with continuous integration systems. Also provided are configuration guidelines for PVS-Studio launching modes.

Introduction

Continuous integration (CI) is the practice of software development implying frequent building and subsequent testing for the most recent versions of designed application. Generally, continuous integration systems interact directly with revision control and allow for a significant increase in integration process's reliability while decreasing its laboriousness by automating entire building-testing phase. PVS-Studio static code analyzer is intended to scan the code for 64-bit portability and parallel errors and thus can be employed at the testing stage during software quality control process.

For the purpose of deployment within continuous integration, PVS-Studio can launch the analysis for its target project in "silent" mode from [command line shell](#). As such, integrating PVS-Studio into periodic build process allows for an effective utilization of analysis's results during collaborate project development process.

Using the "Mark As False Alarm" function during testing automation

While launching from command line shell, PVS-Studio will always analyze the whole target project. The reason for such an approach is that modifications in a single header file can affect multiple cpp files, thus checking only modified files will not be sufficient.

Keep in mind that static analysis will always generate a lot of so called "[false alarms](#)", so to suppress that "noise" it is imperative to employ "Mark as False Alarm" functionality while using PVS-Studio during regular source code verification. Thus PVS-Studio operation can be subdivided into 2 steps: integration and exploitation. During the integration stage developer should review all generated error messages and either correct corresponding code or mark it as false alarm. Afterwards, the analyzer would generate error messages only for newly added code.

Integrating PVS-Studio into build automation process

In the light of PVS-Studio being an extension to Visual Studio IDE, it is possible to integrate the analyzer into any CI system capable of executing Visual Studio builds via command line commands with arguments. Also, direct support for Visual Studio building process is not required from the CI system, the ability to pass commands to operating system's shell and process stdout stream will suffice.

Launching the analyzer from command line is described in detail in this [separate article](#).

The resulting XML-log file can be loaded into Visual Studio Error List (by double-clicking it directly from file manager or through PVS-Studio/Load Analysis Report menu item) and used to navigate through errors in analyzed source code. As the log file itself is not conveniently formatted to be reviewed manually, analyzer will also create the plain text file containing a list of all errors (not marked as false alarms). This file is intuitive and can be conveniently included into continuous integration system's general logs, which in turn can be published, for example, by e-mail to all concerned developers.

In case the verification of all the files from all projects is not an appropriate option, PVS-Studio is able to analyze only the files which were modified during the predefined time interval. This time interval can be defined at the [CommonAnalyzerSettings](#) option page. Alternatively one can also explicitly specify the files for analysis using the batch mode.

Launching the analyzer in batch mode through command line

Launching through command line will by default make the analyzer check all the files included in selected Visual Studio solution. If, however, the verification of only a predefined set of files is required, it is possible to utilize the batch mode which will subject to analysis only files that were explicitly defined by their paths. This launching mode is also mentioned in this [article](#).

Integrating with CruiseControl .NET

PVS-Studio integration with CruiseControl.NET can be achieved by addition of "Executable" task into build project. This task will launch cmd.exe Windows shell and pass the required build arguments into it. Below is the fragment of XML server configuration file, which contains such a task.

```
<exec>
<description>PVS-Studio check solution example</description>
<executable>%CMD_PATH%;</executable>
```

```

<baseDirectory>%PROJECT_ROOT%;</baseDirectory>
<buildArgs>
  <item>
/c PVS-Studio.exe --sln-file VSSol.sln --plog-file result-log.plog
--vcinstalldir "C:\Program Files (x86)\Microsoft Visual Studio
10.0\VC" --platform "Win32" --configuration "Release"
  </item>
</buildArgs>
</exec>

```

The analysis results contained within "result-log.plog.only_new_messages.txt" text file can be published by any available publisher tasks, for example by being moved into common builder directory or by e-mailing build server logs. The inclusion of these results into general builder logs can be performed by such Executable task:

```

<exec>
<description>PVS-Studio load error log</description>
<executable>%CMD_PATH%;</executable>
<baseDirectory>%PROJECT_ROOT%;</baseDirectory>
<buildArgs>
  <item>
/c type result-log.plog.only_new_messages.txt
  </item>
</buildArgs>
<buildTimeoutSeconds>0</buildTimeoutSeconds>
</exec>

```

Integrating with Hudson

Integrating with Hudson can be performed by adding "execute Windows shell command" build step into build project (Job). An example of the command to be added, as follows:

```

PVS-Studio.exe --sln-file VSSol.sln --plog-file result-log.plog
--vcinstalldir "C:\Program Files (x86)\Microsoft Visual Studio
10.0\VC" --platform "Win32" --configuration "Release"

```

Text file holding the list of generated errors (not marked as false alarms) can be included into build server general log by addition of another command:

```
type result-log.plog.only_new_messages.txt
```

Afterwards the resulting build log can be published with the help of any available Hudson publisher tools. For example, to publish results by e-mail, you can use ext-mail plug-in, adding \$BUILD_LOG token into message body:

```
${BUILD_LOG, maxLines=5000}
```

Integrating with Microsoft Team Foundation Server 2005/2008/2010

Using PVS-Studio analyzer together with Team Foundation build server is complicated by the presence of strong integration between Visual Studio projects and Microsoft's own source control system (SCS) thus requiring additional steps compared to other companies' continuous integration systems in case Microsoft Team Foundation SCS is being used. You should also take into account that TFS build server uses MSBuild utility for its builds and the presence of Visual Studio IDE is not required for build server's operations. In the light of PVS-Studio being an extension plug-in to Visual Studio IDE, the access of TFS build server to Visual Studio environment (devenv.exe in particular) with PVS-Studio installed in it is a mandatory requirement.

A typical TFS build process using PVS-Studio analyzer should include the following minimal set of stages:

1. The server downloads required projects' sources into a defined local directory.
2. Program source codes are verified by PVS-Studio analyzer (to do this you should call devenv.exe with **"/command PVStudio.Chechsolution"** command line switch), the results are saved into defined local directory.
3. The server builds the projects using MSBuild.
4. Build and verification results are published into local directory predefined for this particular build.

Team Foundation 2010 Build Server

To integrate PVS-Studio analysis (stage 2) into Team Foundation 2010 server's build process you should add the InvokeProcess build activity to build script defined in xaml file (Edit Build Definition menu item for required build definition, Process tab, Build Process Template field), pass a path to PVS-Studio.exe and this line as an Arguments property:

```
--sln-file VSSol.sln --plog-file result-log.plog --vcinstalldir "C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC" --platform "Win32" --configuration "Release"
```

Team Foundation 2005/2008 Build Server

To integrate PVS-Studio analysis (stage 2) into Team Foundation 2005/2008 server's build process you should modify MSBuild project (file with the proj extension) belonging to the required build definition, adding an Exec build task to it. (The Exec task must be a descendant of one of the Target nodes in which MSBuild tasks are grouped) You should also pass the following string as the command attribute:

```
PVS-Studio.exe --sln-file VSSol.sln --plog-file result-log.plog --vcinstalldir "C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC" --platform "Win32" --configuration "Release"
```

Starting Team Foundation builds in interactive desktop mode.

Team Foundation System build server executes its builds using Local System Account (NT AUTHORITY\SYSTEM) or Local Service account in non-interactive mode by default, i.e. works as Windows service and it is unable to interact with the user's desktop. It is highly recommended to use interactive desktop mode for TFS builds to ensure the correct operation of Visual Studio environment. To launch TFS build in interactive mode you should start Team Foundation Build service from under the account which has privileges to access the desktop (for example your current user account). For Team Foundation 2008 this service can be launched manually (C:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE\PrivateAssemblies\TFSSBuildService.exe). For Team Foundation 2010 the launch properties of this service can be modified using the server configuration utility.

After the first launch of devenv.exe from under the Local System Account (NT AUTHORITY\SYSTEM), Visual Studio will produce a dialog window, allowing you to select one of default environment settings templates. This dialog will also block the operation of PVS-Studio analyzer, thus it should be closed manually at least once, but it is impossible to accomplish for an application which was started in non-interactive mode. Moreover, starting from Windows Vista, the applications launched from under the Local System Account are not allowed to be launched in interactive mode. To bypass this restriction you could use the PsExec utility which is a part of [PSTools](#) utility set. With its help, you can start devenv.exe process from under the Local System Account in interactive mode using the following arguments: **psexec.exe -i -s %devenvPath%**.

The analysis of projects containing Team Foundation Server source control.

If Microsoft Team Foundation (TFS versions 2008/2010) is used as source control system, this modus operandi makes it impossible to open and analyze Visual Studio solution files that contain TFS source control meta-information because the dialog windows produced by the IDE will block the operations of PVS-Studio extension. PVS-Studio does not support command line launched verification of projects containing TFS 2008/2010 source control meta-information.

To begin analyzing Visual C++ projects using PVS-Studio analyzer from Team Foundation System build server which utilizes Team System source control, you should erase all TFS source control metainformation from sln and projects (vcxproj) files. You can accomplish this without interrupting build process and in fully automated mode with the help of **TFSRipper** command line utility. To accomplish this you should add the call to this utility before and after calling devenv.exe, also using InvokeProcess build activity. The utility should receive the **%LocalSourcePath%** – path to sources to be analyzed downloaded by the build server – as argument while it is launched the first time, and **%LocalSourcePath% \restore** argument while it is launched second time. The utility will process all sln and vcxproj files, deleting the source control metainformation and creating the copies of these files beforehand (with the pvsbak extension), and it will restore all files from earlier created copies after being called second time with **/restore** argument.

Predefined PVS_STUDIO macro

Among the numerous filtration and message suppression methods of PVS-Studio analyzer is the PVS_STUDIO predefined macro. Its purpose is to prevent the marked source code from being analyzed. For example, let the analyzer produce the message for the following code:

```
int rawArray[5];
rawArray[-1] = 0;
```

However, if you will 'wrap' it using this macro the message will not be generated.

```
int rawArray[5];
#ifdef PVS_STUDIO
    rawArray[-1] = 0;
#endif
```

The PVS_STUDIO macro is automatically inserted while checking the code from Visual Studio. But if you are using PVS-Studio [from the command line](#), the macro will not be passed by default to the analyzer and this should be done manually.

PVS-Studio FAQ

[Message "Files with C or C++ source code for analysis not found." appears while checking a group of projects or a separate C or C++ project.](#)

[I get a huge amount of messages "V001. A code fragment from 'file' cannot be analyzed", the analyzer won't work at all.](#)

[Why does multi-thread check \(ThreadCount setting\) sometimes work slower than single-thread check?](#)

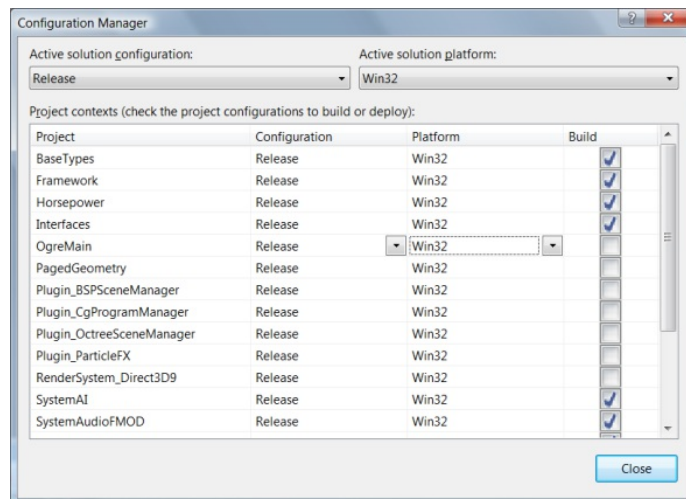
[Why does the tool have only Windows version? What are the difficulties of verifying the text files \(containing source code\) produced by different compilers?](#)

[I don't have the PVS-Studio menu in Visual Studio Express Edition after installation. Why?](#)

Message "Files with C or C++ source code for analysis not found." appears while checking a group of projects or a separate C or C++ project.

It means that projects are disabled in the common build through the Configuration Manager window of the Visual Studio environment.

To perform correct analysis of C or C++ projects with the PVS-Studio static analyzer, these projects must be compilable in Visual C++ and buildable without errors. That is why PVS-Studio performs analysis only of those projects that are included in the common build when checking a group of projects or a separate project.



Projects that are not included into the build will be skipped. If none of the existing projects are included into the build or if you choose to check a project not included into the build, the analyzer will show the message "Files with C or C++ source code for analysis not found" and analysis will not be launched. You may see what projects are enabled and disabled in the common build with the help of the Configuration Manager window for the current Visual Studio solution.

I get a huge amount of messages "V001. A code fragment from 'file' cannot be analyzed", the analyzer won't work at all.

The analyzer can sometimes fail to analyze a source code file. It is not always a fault of the analyzer; please read about the reasons for this behavior in the documentation for [V001](#). Whatever the reason for displaying the V001 message, it is not crucial. As a rule, incomplete parse of a file is not very significant from the viewpoint of analysis. PVS-Studio simply skips a function/class with an error and continues analyzing the file. Only a very small code fragment remains unchecked.

Why does multi-thread check (ThreadCount setting) sometimes work slower than single-thread check?

The PVS-Studio requires rather large amount of memory. Perhaps the reason is that the analyzer is a lot and therefore everything slows down. Each running instance of the analyzer requires approx. 1.5 Gb of RAM. So, for instance, the configuration of 4 cores and 2 Gbytes might appear much slower than a system

with 2 cores and 4 Gbytes of memory.

You may set the number of cores to be used during the analysis in the PVS-Studio's settings (the ThreadCount option).

Why does the tool have only Windows version? What are the difficulties of verifying the text files (containing source code) produced by different compilers?

Technically, it is easy to port the code of PVS-Studio.exe to a different platform since there are almost no Windows-specific functions being used there.

To create PVS-Studio for a different operating system, we must provide at least the same quality of the tool on a new platform as on the Windows-platform. To provide the high quality of PVS-Studio, we use a lot of various types of tests: static analysis, unit-tests, functional tests, UI tests, etc. Only after passing all these tests we can be sure that the analyzer behaves in the same way on a different platform as on the Windows-platform. Testing the analyzer takes about 30% of the general price of its development.

Also note that the files containing source code produced by the Visual C++ compiler contain particular constructs specific to this compiler. PVS-Studio supports them. Other compilers (for instance, GCC) also have their own specific constructs that we will have to support. This is an unapparent yet very large amount of work here. The fact that the '___restrict' key word is used 10 000 times more rarely than 'for' does not mean that we must not provide support for it. Many programmers even do not think about such things as __w64, __noop, __if_exists, __int3264, __uidof, __based, __LPREFIX and so on and so forth.

Finally, sets of include files used in different systems are also different. If PVS-Studio supports include files of Visual C++, it does not necessarily mean that it will successfully handle GCC's include files.

Let's summarize everything written here. We can develop PVS-Studio for a different platform, but it is a serious task we are not intending to get involved into. If our customer wants to have PVS-Studio for some other platform, we can develop it at his expense.

I don't have the PVS-Studio menu in Visual Studio Express Edition after installation. Why?

No version of Visual Studio Express Edition supports extension packages - this is a restriction of this particular edition of Visual Studio, so there is no fault on PVS-Studio's side.

Tips on speeding up PVS-Studio

[Use a multi-core computer with a large amount of memory.](#)

[Use an SSD both for the system and the project to be analyzed](#)

[Configure \(or turn off\) your antivirus](#)

[If possible, use Clang as the preprocessor instead of Visual C++ \(it can be chosen in the PVS-Studio settings\)](#)

[Exclude libraries you don't need from analysis \(can be set in the PVS-Studio settings\)](#)

[Conclusion](#)

Any static code analyzer works slower than a compiler. It is determined by the fact that the compiler must work very quickly, though to the detriment of analysis depth. Static analyzers have to store the [parse tree](#) to be able to gather more information. Storing the parse tree increases memory consumption, while a lot of checks turn the tree traverse operation into a resource-intensive and slow process. Well, actually it all is not so much crucial, since analysis is a rarer operation than compilation and users can wait a bit. However, we always want our tools to work faster. The article contains tips on how to significantly increase PVS-Studio's speed.

At first let's enumerate all the recommendations so that users learn right away how they can make the analyzer work faster:

1. Use a multi-core computer with a large amount of memory.
2. Use an SSD both for the system and the project to be analyzed.
3. Configure (or turn off) your antivirus.
4. If possible, use Clang as the preprocessor instead of Visual C++ (it can be chosen in the PVS-Studio settings).
5. Exclude libraries you don't need from analysis (can be set in the PVS-Studio settings).

Let's consider all these recommendations in detail, explaining why they allow the tool to work faster.

Use a multi-core computer with a large amount of memory.

PVS-Studio has been supporting multi-thread operation for a long time already (starting with version 3.00 released in 2009). Parallelization is performed at the file level. If analysis is run on four cores, the tool is checking four files at a time. This level of parallelism enables you to get a significant performance boost. Judging by our measurements, there is a marked difference between the four-thread and one-thread analysis modes of test projects. One-thread analysis takes 3 hours and 11 minutes, while four-thread analysis takes 1 hour and 11 minutes (these data were obtained on a four-core computer with 8 Gbytes of memory). That is, the difference is 2.7 times.

It is recommended that you have at least one Gbyte of memory for each analyzer's thread. Otherwise (when there are many threads and little memory), the swap file will be used, which will slow down the analysis process. If necessary, you may restrict the number of the analyzer's threads in the PVS-Studio settings: Options -> Common Analyzer Settings -> Thread Count ([documentation](#)). By default, the number of threads launched corresponds to the number of cores available in the system.

We recommend that you use a computer with four cores and eight Gbytes of memory or better.

Use an SSD both for the system and the project to be analyzed

Strange as it may seem, a slow hard disk is a bottleneck for the code analyzer's work. But we must explain the mechanism of its work for you to understand why it is so. To analyze a file, the tool must first preprocess it, i.e. expand all the #define's, include all the #include's and so on. The preprocessed file has an average size of 10 Mbytes and is written on the disk into the project folder. Only then the analyzer reads and parses it. The file's size is growing because of that very inclusion of the contents of the #include-files read from the system folders.

I can't give exact results of measuring the influence of an SSD on the analysis speed because you have to test absolutely identical computers with only hard disks different. But visually the speed-up is great.

Configure (or turn off) your antivirus

Judging by the character of its work, the analyzer is a complex and suspicious program from the viewpoint of an antivirus. Let's specify right away that we don't mean that the analyzer is recognized as a virus - we check this regularly. Besides, we use a code certificate signature. Let's go back to description of the code analyzer's work.

For each file being analyzed a separate analyzer's process is run (the PVS-Studio.exe module). If a project contains 3000 files, the same number of PVS-Studio.exe's instances will be launched. PVS-Studio.exe calls Visual C++ environment variable setting (files vcvars*.bat) for its purposes. It also creates a lot of

preprocessed files (*.i) (one for each file being compiled) for the time of its work. Auxiliary command (.cmd) files are being used.

Although all these actions are not a virus activity, it still makes any antivirus spend many resources on meaningless check of the same things.

We recommend that you add the following exceptions in the antivirus's settings:

1. Do not scan system folders with Visual Studio:
 - a. C:\Program Files (x86)\Microsoft Visual Studio 8
 - b. C:\Program Files (x86)\Microsoft Visual Studio 9.0
 - c. C:\Program Files (x86)\Microsoft Visual Studio 10.0
 - d. etc.
2. Do not scan the PVS-Studio folder:
 - a. C:\Program Files (x86)\PVS-Studio
3. Do not scan the project folder:
 - a. For example, C:\Users\UserName\Documents\MyProject
4. Do not scan Visual Studio .exe files:
 - a. C:\Program Files (x86)\Microsoft Visual Studio 8\Common7\IDE\devenv.exe
 - b. C:\Program Files (x86)\Microsoft Visual Studio 9.0\Common7\IDE\devenv.exe
 - c. C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\devenv.exe
 - d. etc.
5. Do not scan the cl.exe compiler's .exe files (of different versions):
 - a. C:\Program Files (x86)\Microsoft Visual Studio 8\VC\bin\cl.exe
 - b. C:\Program Files (x86)\Microsoft Visual Studio 8\VC\bin\x86_amd64\cl.exe
 - c. C:\Program Files (x86)\Microsoft Visual Studio 8\VC\bin\amd64\cl.exe
 - d. C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\bin\cl.exe
 - e. C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\bin\x86_amd64\cl.exe
 - f. C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\bin\amd64\cl.exe
 - g. C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\cl.exe
 - h. C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\x86_amd64\cl.exe
 - i. C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\amd64\cl.exe
 - j. etc.
6. Do not scan PVS-Studio and Clang .exe files (of different versions):
 - a. C:\Program Files (x86)\PVS-Studio\x86\PVS-Studio.exe
 - b. C:\Program Files (x86)\PVS-Studio\x86\clang.exe
 - c. C:\Program Files (x86)\PVS-Studio\x64\PVS-Studio.exe
 - d. C:\Program Files (x86)\PVS-Studio\x64\clang.exe

Perhaps this list is too excessive but we give it in this complete form so that you know regardless of a particular antivirus what files and processes do not need to be scanned.

Sometimes there can be no antivirus at all (for instance, on a computer intended specially to build code and run a code analyzer). In this case the speed will be the highest. Even if you have specified the above mentioned exceptions in your antivirus, it still will spend some time on scanning them.

Our test measurements show that an aggressive antivirus might slow down the code analyzer's work twice or more.

If possible, use Clang as the preprocessor instead of Visual C++ (it can be chosen in the PVS-Studio settings)

PVS-Studio exploits an external preprocessor. Earlier we used only one preprocessor by Microsoft Visual C++. In PVS-Studio 4.50 we added support of another preprocessor Clang that works much faster and doesn't have certain weak points of the Microsoft preprocessor (although it does have its own). However, using the Clang preprocessor will in most cases make the analyzer's work faster 1.5-1.7 times.

But there is one thing you should remember. You may specify the preprocessor to be used in the PVS-Studio settings: Options -> Common Analyzer Settings -> Preprocessor ([documentation](#)). There are three alternatives available: VisualCPP, Clang and VisualCPPAfterClang. The first two are obvious. What the third alternative is concerned, it means that Clang will be used first and if some errors occur during preprocessing, the file will be then preprocessed again by Visual C++. It is this option (VisualCPPAfterClang) which is chosen by default.

If your project is analyzed with Clang without any problems, you may use the default option VisualCPPAfterClang or Clang - it doesn't matter. But if your project can be checked only with Visual C++, you'd better specify this option so that the analyzer doesn't launch Clang in vain trying to preprocess your files.

Exclude libraries you don't need from analysis (can be set in the PVS-Studio settings)

Any large software project uses a lot of third-party libraries such as zlib, libjpeg, Boost, etc. Sometimes these libraries are built separately, and in this case the main project has access only to the header and library (lib) files. And sometimes libraries are integrated very firmly into a project and virtually become part of it. In this case the main project is compiled together with the code files of these libraries.

The PVS-Studio analyzer can be set to not check code of third-party libraries: even if there are some errors there, you most likely won't fix them. But if you exclude such folders from analysis, you can significantly enhance the analysis speed in general.

It is also reasonable to exclude code that surely will not be changed for a long time from analysis.

To exclude some folders or separate files from analysis use the PVS-Studio settings -> Don't Check Files ([documentation](#)).

To exclude folders you can specify in the folder list either one common folder like c:\external-libs, or list some of the folders: c:\external-libs\zlib, c:\external-libs\libjpeg, etc. You can specify a full path, a relative path or a mask. For example, you can just specify zlib and libjpeg in the folder list - this will be automatically considered as a folder with mask *zlib* and *libjpeg*. To learn more, please see the [documentation](#).

Conclusion

Let's once again list the methods of speeding up PVS-Studio:

1. Use a multi-core computer with a large amount of memory.
2. Use an SSD both for the system and the project to be analyzed.
3. Configure (or turn off) your antivirus.
4. If possible, use Clang as the preprocessor instead of Visual C++ (it can be chosen in the PVS-Studio settings).
5. Exclude libraries you don't need from analysis (can be set in the PVS-Studio settings).

The greatest effect can be achieved when applying all these recommendations simultaneously.

PVS-Studio menu commands

[Check Current File](#)

[Check Current Project](#)
[Check Selected item\(s\)](#)
[Check Solution](#)
[Incremental Analysis after Build](#)
[Disable incremental analysis until IDE restart](#)
[Open/Save](#)
[Open Analysis Report](#)
[Save Analysis Report As...](#)
[Save Analysis Report](#)
[Export current configuration to file...](#)
[Import current configuration from file...](#)
[Reset to default configuration](#)
[Reset detectable errors to default](#)
[Show PVS-Studio Output window](#)
[Find in PVS-Studio output...](#)
[Help](#)
[Open PVS-Studio Documentation \(PDF\)](#)
[Open PVS-Studio Documentation \(html, online\)](#)
[Open OmniSample.zip examples](#)
[Request support via website](#)
[Check for updates](#)
[Enter registration information...](#)
[Options](#)

Check Current File

This command launches PVS-Studio analysis on a file that is currently opened in the Visual Studio code editor window for the selected build configuration and platform. If there are no files opened in the editor then the verification will be started on the file that is currently selected in the Solution Explorer window. It should be noted that this command is primarily designed to verify the code that is opened in the IDE code editor and the 'Check Selected item(s)' menu command should be used for the convenient verification of selected items (single or multiple ones) from the Solution Explorer

Check Current Project

This command launches PVS-Studio analysis on all the files of the project that is currently selected in the IDE 'Solution Explorer' window for the selected build configuration and platform. This command is intended to check only the single current project, the 'Check Selected item(s)' menu command should be used to verify several projects at once.

Check Selected item(s)

This command launches PVS-Studio analysis on all the items currently selected in the 'Solution Explorer' IDE window for the selected build configuration and platform. Any single C/C++ file, entire Visual C++ projects and whole solution folders could be selected for analysis, but the 'Check Solution' menu command is a more convenient way to start the verification for all the items from within the Solution Explorer tree.

Check Solution

This command starts PVS-Studio analysis on all C/C++ files belonging to the solution that is currently opened in Visual Studio IDE for the selected build configuration and platform, i.e. on all the projects from the 'Solution Explorer' window. It should be noted that analysis will not be started for the projects which are excluded from the build in the Configuration Manager settings for current build configuration.

Incremental Analysis after Build

The 'Incremental Analysis after Build' command allows you to specify the mode (turn on/off) for PVS-Studio incremental analysis. When enabled PVS-Studio IDE extension will monitor the building of Visual C++ projects and individual C/C++ files so that the files that were modified since the last successful build will be automatically verified.

Disable incremental analysis until IDE restart

When using the PVS-Studio incremental analysis ('Incremental Analysis after Build' command) this command allows it to be temporarily disabled for the current Visual Studio IDE session, i.e. incremental analysis will remain enabled after the IDE is restarted. This menu item is always unchecked by default after the IDE is restarted. The incremental analysis also can be temporarily turned off through the context menu in Windows notification area (PVS-Studio tray icon).

Open/Save

The Open/Save submenu contains several groups of subcommands which are responsible for handling PVS-Studio log files (files with plog extension) and analyzer's configuration file (settings.xml).

Open Analysis Report

The 'Open Analysis Report' command allows you to load a previously saved analyzer's log into PVS-Studio IDE Output window. This command permits for a standard XML log (plog extension) to be opened as well as the unparsed PVS-Studio output in case the analyzer was started from a command line in IDE independent mode (the 'unparsed output' type should be selected in the standard file opening dialog).

Save Analysis Report As...

This allows the current analyzer results from the PVS-Studio Output window to be saved either as a standard XML file (plog type) or as a plain text file, the type could be specified through the drop-down menu in the standard Windows file saving dialog.

Save Analysis Report

This command allows the current analyzer output to be saved. In case the output was generated by loading the existing PVS-Studio log file or it was already saved once during the current Visual Studio session, the initial file will be replaced. Otherwise the standard file saving Windows dialog will be displayed (as with the 'Save Analysis Report As...' command)

Export current configuration to file...

The command allows for the current analyzer's settings to be exported into an external XML file. The settings used by the analyzer are stored in the Settings.xml file located in the %AppData%/PVS-Studio directory.

Import current configuration from file...

This command allows the analyzer's settings to be imported from an external PVS-Studio XML settings file. These imported settings will overwrite the current analyzer's settings which are stored in the Settings.xml file located in the %AppData%/PVS-Studio directory. The current settings could also be exported into an external XML file with the help of the 'Export current configuration to file...' command.

Reset to default configuration

This command allows you to reset the analyzer's settings to default values. Using this command will not cause the registration information to be lost. It should be noted that in order to reset the display of various error types to default (the Detectable errors menu) it is far more convenient to use the 'Reset detectable errors to default' menu command as it will leave other suppression filters unaffected.

Reset detectable errors to default

This allows you to reset the 'Detectable errors' filters to default values, in which the most important diagnostics are enabled. To fully reset the PVS-Studio settings (including all of the message suppression filters) the 'Reset to default configuration' menu command should be used instead.

Show PVS-Studio Output window

This command either opens the PVS-Studio Output IDE window or sets a focus to it if this window had already been opened before.

Find in PVS-Studio output...

This command opens the 'Find in PVS-Studio Output' IDE window or sets a focus to it if this window had already been opened before. This window could also be opened using the IDE 'Find in PVS-Studio output...' tool panel button in the 'PVS-Studio' output window. The 'Find in PVS-Studio output' window allows you to perform a text search in the analyzer's output displayed inside the PVS-Studio IDE window.

Help

The Help submenu contains commands intended to provide user support, access to the analyzer's documentation and the analyzer's registration settings.

Open PVS-Studio Documentation (PDF)

The command opens a unified PDF file which contains analyzer's full documentation. This documentation is also available on our website through the Open PVS-Studio Documentation (html, online) menu command.

Open PVS-Studio Documentation (html, online)

This command opens the contents of analyzer's documentation available on viva64.com website. The local version of this content is available in the form of a single PDF file through the Open PVS-Studio Documentation (PDF) menu command.

Open OmniSample.zip examples

Allows the OmniSample.zip archive to be opened. It contains examples of various issues detectable by the PVS-Studio analyzer (the OmniSample projects are for Visual Studio 2005/2008/2010 IDE)

Request support via website

This command opens the page of viva64.com website allowing you to request a technical user support for the PVS-Studio analyzer.

Check for updates

Allows you to check the availability of updates for PVS-Studio analyzer. It should be noted that the Visual Studio IDE extension by default checks for updates before starting the analysis of the source code (the CheckForNewVerions settings item at the PVS-Studio->Common Analyzer Settings page)

Enter registration information...

Allows you to enter the registration information for the PVS-Studio analyzer. By default the analyzer operates in a trial mode which does not require any additional registration information to be specified. The registration information could also be entered by opening the Registration option's page using the Options or Tools->Options menu commands. The restart of the IDE is not required to change the current registration state.

Options

This command opens PVS-Studio options page that is integrated into the IDE common settings. It's worth noting that PVS-Studio page could also be accessed via the standard IDE settings dialog at the Tools->Options->PVS-Studio.

Settings: General

When developing PVS-Studio we assigned primary importance to the simplicity of use. We took into account our experience of working with traditional lint-like code analyzers. And that is why one of the main advantages of PVS-Studio over other code analyzers is that you can start using it immediately. Besides, PVS-Studio has been designed in such a way that the developer using the analyzer would not have to set it up at all. We managed to solve this task: a developer has a powerful code analyzer which you need not to set up at the first launch.

But you should understand that the code analyzer is a powerful tool which needs competent use. It is this competent use of the analyzer (thanks to the settings system) that allows you to achieve significant results. Operation of the code analyzer implies that there should be a tool (a program) which performs routine work of searching potentially unsafe constructions in code and a master (a developer) who can make decisions on the basis of what he knows about the project being verified. Thus, for example, the developer can inform the analyzer that:

- some error types are not important for analysis and do not need to be shown (with the help of settings of [Settings: Detectable Errors](#));

- the project does not operate with large data arrays of some types (by defining it in the settings, [Viva64](#));
- the project does not contain incorrect type conversions (by disabling the corresponding diagnostic messages, [Settings: Detectable Errors](#));

Correct setting of these parameters can greatly reduce the number of diagnostic messages produced by the code analyzer. It means that if the developer helps the analyzer and gives it some additional information by using the settings, the analyzer will in its turn reduce the number of places in the code which the developer must pay attention to when examining the analysis results.

Setting up PVS-Studio is performed through a mechanism standard for Microsoft Visual Studio - "Options" command in "Tools" menu. When selecting this command you will see the dialogue of Microsoft Visual Studio setting and one of its menu items will be PVS-Studio.

Each setting dialogue is described in PVS-Studio documentation.

Settings: Common Analyzer Settings

[Check For New Versions](#)

[Check only Files Modified In](#)

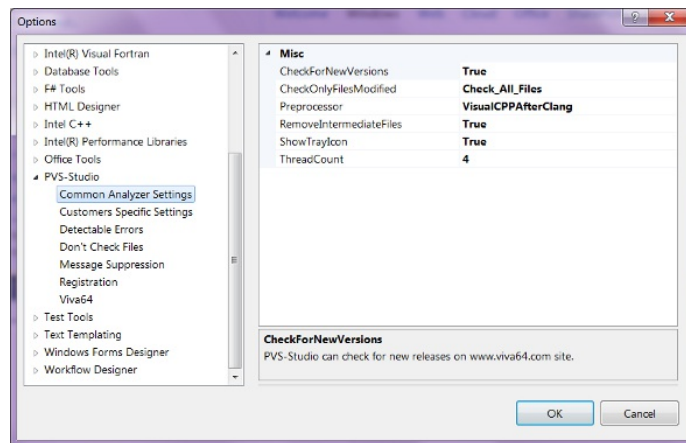
[Preprocessor](#)

[Remove Intermediate Files](#)

[Show Tray Icon](#)

[Thread Count](#)

The tab of the analyzer's general settings displays the settings which do not depend on the particular analysis unit being used.



Check For New Versions

The analyzer can automatically check for updates on www.viva64.com site. It uses our update module.

If the CheckForNewVersions option is set to True, a special text file is downloaded from www.viva64.com site when you launch code checking (the commands Check Current File, Check Current Project, Check Solution in PVS-Studio menu). This file contains the number of the latest PVS-Studio version available on the site. If the version on the site is newer than the version installed on the user computer, the user will be asked for a permission to update the program. If the user agrees, a special separate application PVS-Studio-Updater will be launched that will automatically download and install the new PVS-Studio distribution kit. If the option CheckForNewVersions is set to False, it will not check for the updates.

Check only Files Modified In

This field allows you to define the time interval in which the presence of modifications in analyzed files will be controlled using "Date Modified" file attribute. With that, only the files with "Date Modified" attribute not exceeding this interval will be sent for analysis, that is, only the recently modified files, and only the recently added or modified code will be verified.

Controlling file modifications will solve the potential problem of inadmissible duration of whole project's verification and nightly builds taking too long, but it still holds several limitations:

1. Modification of header files will lead to the exclusion from verification of c/cpp files in which these headers are included in case these c/cpp files are not being modified themselves.
2. During a full checkout from version control's repository the "Date Modified" attribute could be rewritten for all files by system with a date of the checkout operation itself, i.e. the modification time for every file will be the same. The same situation can theoretically arise from simple file copy as well.

In other words, this approach would allow for verification of "all files modified today".

Preprocessor

An external preprocessor is being utilized to preprocess files with PVS-Studio. It is only Microsoft Visual C++ preprocessor that had been employed for this task in the past. But in 4.50 version of PVS-Studio the support for the Clang preprocessor had been added, as its performance is significantly higher and it lacks some of the Microsoft's preprocessor shortcomings (although it also possesses issues of its own). Still, the utilization of Clang preprocessor provides an increase of operational performance by 1.5-1.7 times in most cases.

However there is an aspect that should be considered. The preprocessor to be used can be specified from within the PVS-Studio Options -> Common Analyzer Settings -> Preprocessor field. The available options are: VisualCPP, Clang and VisualCPPAfterClang. The first two of these are self evident. The third one indicates that Clang will be used at first, and if preprocessing errors are encountered, the same file will be preprocessed by the Visual C++ preprocessor instead. This option is a default one (VisualCPPAfterClang).

Remove Intermediate Files

The analyzer creates a lot of temporary command files for its operation to launch the analysis unit itself, to perform preprocessing and to manage the whole process of analysis. Such files are created for each project file being analyzed. Usually they are not of interest for a user and are removed after the analysis process. But in some cases it can be useful to look through these files. So you can indicate to the analyzer not to remove them. In this case you can launch the analyzer outside Visual Studio environment from the command line.

Show Tray Icon

This setting allows you to control the notifications of PVS-Studio analyzer operations. In case PVS-Studio output window contains error messages after performing the analysis (the messages potentially can be concealed by various filters as false alarms, by the names of files being verified and so on; such messages will not be present in PVS-Studio window), the analyzer will inform you about their presence with popup message in the Windows notification area (System tray). Single mouse click on this message or PVS-Studio tray icon will open the output window containing the messages which were found by the analyzer.

Thread Count

Analysis of files is performed faster on multi-core computers. Thus, on a 4-core computer the analyzer can use all the four cores for its operation. But if for some reasons you need to limit the number of cores being used you can do this by selecting the needed number.

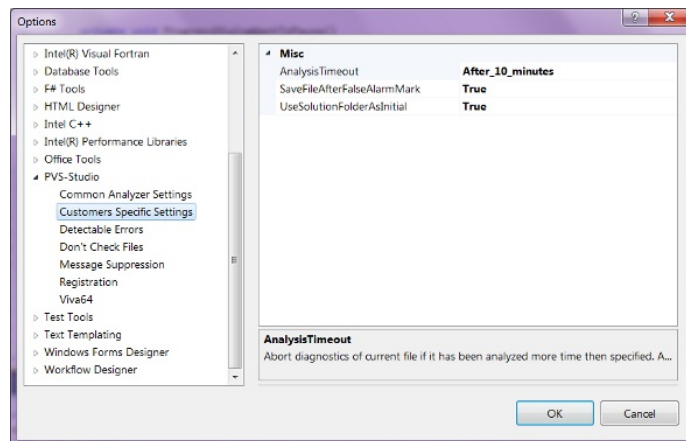
Settings: Customers Specific Settings

[Analysis Timeout](#)

[Save File After False Alarm Mark](#)

[Use Solution Folder As Initial](#)

The "Customers Specific Settings" tab contains settings intended for integration of the tool into development environments of our users owning the commercial license for the analyzer.



Analysis Timeout

This setting allows you to set the time limit, by reaching which the analysis of individual files will be aborted with [V006. File cannot be processed. Analysis aborted by timeout](#) error, or to completely disable analysis termination by timeout. We strongly advise you to consult the description of the error cited above before modifying this setting. The timeout is often caused by the shortage of RAM. In such a case it is reasonable not to increase the time but to decrease the number of parallel threads being utilized. This can lead to a substantial increase in performance in case the processor possesses numerous cores but RAM capacity is insufficient.

Save File After False Alarm Mark

Marking the message as False alarm requires the modification of source code files. By default the analyzer will save each source code file after making every such mark. However, if such frequent savings of files are undesirable (for example if the files are being stored on different machine in LAN), they can be disabled using this setting.

Exercise caution while modifying this setting because the not saving the files after marking them with false alarms can lead to a loss of work in case of Visual Studio IDE being closed.

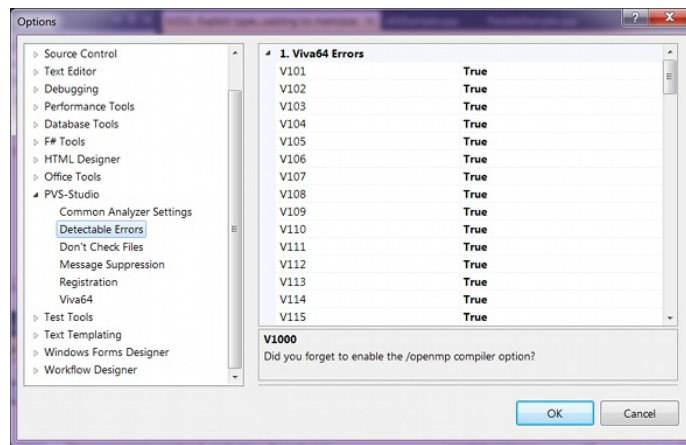
Use Solution Folder As Initial

By default PVS-Studio offers saving report file (.log) inside the same folder as the current solution file.

Modifying this setting allows you to restore the usual behavior of Windows file dialogs, i.e. the dialog will remember the last folder that was opened in it and will use this folder as initial.

Settings: Detectable Errors

In the tab of detectable errors you can indicate the codes of those errors which you do not want to see in the analysis report.

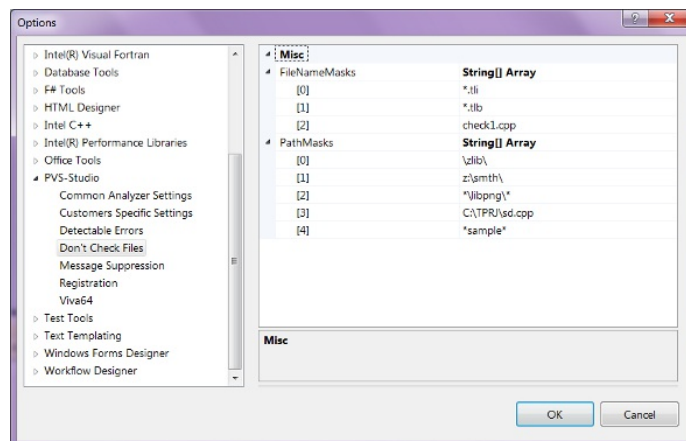


The analyzer detects all the errors supported by this version independently from the settings defined in this settings tab. But sometimes it is needed to omit errors with particular codes in the report. For example, if you are sure that you are not interested in errors relating to explicit type conversion (codes V201, V202, V203) you may skip them in the report. For this you need just to set False value opposite each of these errors' codes.

Please note: when changing the list of errors shown you don't need to restart the project's analysis. The analyzer always detects all the errors and showing particular errors is defined in this settings tab. When enabling diagnosis of errors being disabled earlier, they will appear in the report at once and you won't need to start analysis of the whole project again.

Settings: Don't Check Files

You may specify file masks to exclude some of the files or folders from analysis on the tab "Don't Check Files". The analyzer will not check those files that meet the masks' conditions.



Using this technique, you may, for instance, exclude autogenerated files from the analysis. Besides, you may define the files to be excluded from analysis by the name of the folder they are located in.

A mask is defined with the help of wild card match types. The '*' (any number of any characters) wild card can be used, the '?' symbol is not supported.

The case of a character is irrelevant. The '*' wildcard character could only be inserted at the beginning or at the end of the mask, therefore the masks of the 'a*b' kind are not supported. After exclusion masks were specified, the messages from files corresponding to these masks should disappear from PVS-Studio Output window, and the next time then analysis is started these files will be excluded from it. Thereby the total time of the entire project's analysis could be substantially decreased by excluding files and directories with these masks.

2 types of masks could be specified: the Path masks and the File name masks. The masks specified from within the FileNameMasks list are used to filter messages by the names of the corresponding files only and ignoring these files' location. The masks from the PathMasks list, on the other hand, are used to filter messages by taking into account their location within the filesystem on the disk and could be used to suppress diagnostics either from the single file or even from the whole directories and subdirectories. To filter the messages from one specific file, the full path to it should be added to the PathMasks list, but to filter files sharing the same name (or with the names complying to the wildcard mask), such names or masks should be inserted into the FileNameMask list.

Valid masks examples for the FileNameMask property:

*ex.c — all files with the names ending with "ex" characters and "c" extension will be excluded.

*.cpp — all files possessing the "cpp" extension will be excluded

stdafx.cpp — every file possessing such name will be excluded from analysis regardless of its location within the filesystem

Valid masks examples for the PathMasks property:

c:\Libs\ — all files located in this directory and its subdirectories will be excluded

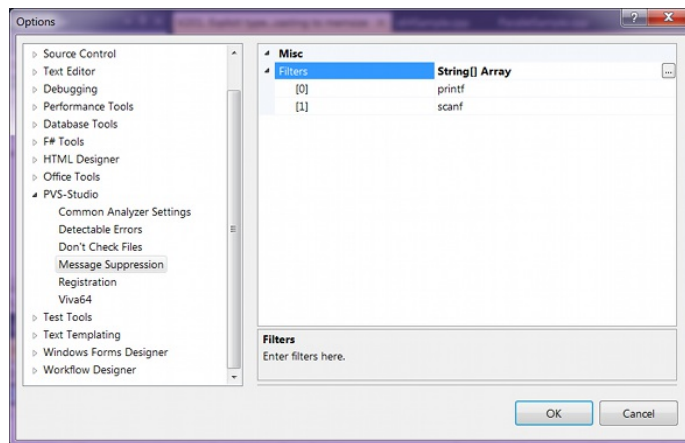
\Libs or *Libs* — all files located in the directories with path containing the Libs subdirectory will be excluded.

Libs or *Libs* — the files possessing within their paths the subdirectory with the 'Libs' chars in its name will be excluded. Also the files with names containing the 'libs' characters will be excluded as well, for example 'c:\project\mylibs.cpp.' To avoid confusion we advise you always to specify folders with slash separators.

c:\proj\includes.cpp — a single file located in the c:\proj\ folder with the specified name will be excluded from the analysis.

Settings: Message Suppression

In the tab of suppression particular messages you can set filtration of errors by the text they contain.



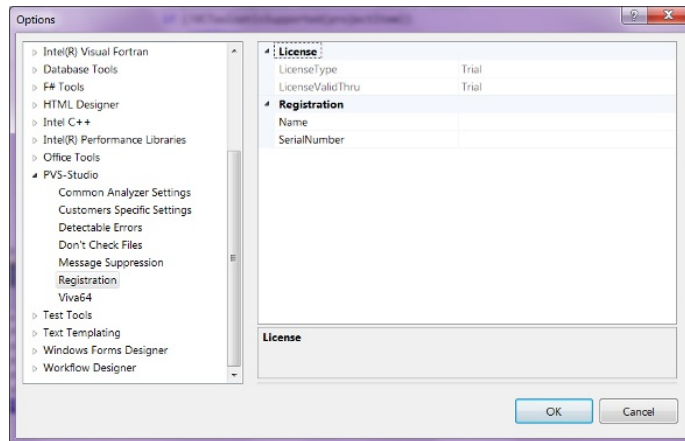
When it's necessary you may hide from analyzer's report the diagnosed errors containing particular words or phrases. For example, if the report contains errors in which names of printf and scanf functions are indicated and you consider that there can be no errors relating to them just add these two words using the message suppression editor.

Please note! When changing the list of the hidden messages you don't need to restart analysis of the project. The analyzer always generates all the diagnostic messages and the display of various messages is managed with the help of this settings tab. When modifying message filters the changes will immediately appear in the report and you won't need to launch analysis of the whole project again.

Settings: Registration

Open PVS-Studio settings page. (PVS-Studio Menu -> Options...).

In the registration tab the licensing information is entered.



After purchasing the analyzer you receive registration information: the name and the serial number. These data must be entered in this tab. In the Application field the licensing mode will be indicated.

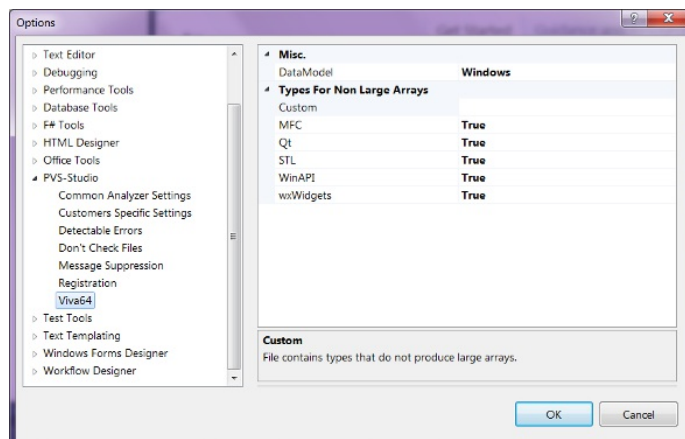
Information on the licensing conditions is located in the [ordering page](#) on site and in the "Registration" section of the Help system.

Settings: Viva64

[DataModel](#)

[TypesForNonLargeArrays](#)

In Viva64 tab there are settings characteristic only of the 64-bit code analysis unit.



DataModel

With the help of this setting, the data model used for code analysis can be changed. For traditional Windows applications verification, LLP64 data model should be used (this mode is selected by default); for Linux applications verification emulation use LP64 data model. Do not change the data model if you are not sure that you need to do that.

From the point of view of 64-bit applications development, Windows and Linux operational systems differ in the data model being used. This is the correlation between the sizes of base data types on a particular system. In 64-bit Windows version, LLP64 data model is used. In such data model, types have the following sizes: int - 32 bit, long - 32 bit, long long - 64 bit, size_t - 64 bit, pointer- 64 bit. In 64-bit Linux version, LP64 data model is used. Data types sizes are the following in it: int - 32 bit, long - 64 bit, long long - 64 bit, size_t - 64 bit, pointer- 64 bit. In 64-bit Linux and Windows systems, such types as long double also differ, however this is not important within the scope of the current discussion. As we can see, the main difference lies in the size of long type - on 64-bit Windows system it is 32 bits, while on 64-bit Linux system it is 64 bits.

The main data models are given in figure.

short	int	long	ptr	long long	Label	Examples
...	16	...	16	...	IP16	PDP-11 Unix (1973)
16	16	32	16	...	IP16L32	PDP-11 Unix (1977); multiple instructions for long
16	16	32	32	...	I16LP32	MC68000 (1982); Apple Macintosh 68K; Microsoft operation systems (plus extras for x86 segments)
16	32	32	32	...	ILP32	IBM 370; VAX Unix; many workstation
16	32	32	32	64	ILP32LL or ILP32LL64	Microsoft Win32; Amdahl; Convex; 1990 Unix systems; Like IP16L32, for same reason; multiple instructions for long long
16	32	32	64	64	LLP64 or IL32LLP64 or P64	64-bit systems Microsoft Win64 (X64 / IA64) Most Unix systems (Linux, Solaris, DEC OSF/1 Alpha, SGI Irix, HP UX 11) HAL; logical analog of ILP32 UNICOS
16	32	64	64	64	LP64 or I32LP64	
16	64	64	64	64	ILP64	
64	64	64	64	64	SILP64	

By default, PVS-Studio code analyzer, being designed for work in Windows environment, is oriented to LLP64 data model. Roughly speaking, this means that it searches incorrect shared use of size_t types and int types. For correctness code verification in 64-bit Linux environment, incorrect shared use of long types and int types should also be searched for.

The possibility of search of incorrect shared use also of long types is activated after the selection of data model LP64. These are special operation conditions, so-called "Linux-like mode". In it, the analyzer of 64-bit problems will work the way it would work in Linux environment. This will allow to partially verify the code of applications which work on Linux, with the help of PVS-Studio, on a Windows system in Visual Studio environment.

Thus, in order to verify a 64-bit Linux application, you should have it able to compile into a Visual Studio version. That means that such verification is quite reasonable for cross-platform applications, which are built both in Windows and in Linux.

Of course, verification of an application on the issue of 64-bit code correctness in the mode of Linux environment simulation cannot be considered full and ensure functioning capability in real Linux world. Besides, additional false operations can occur. However, in a number of cases, with the help of PVS-Studio, this variant of verification allows to find large quantities of errors which will occur on 64-bit Linux systems.

In such an approach, a bottleneck is first of all those code areas which are not built under Windows. E.g., these can be interface program parts. It is impossible to verify them on Windows because they are impossible to build. But, the program kernel, for example, which works both on Windows and on Linux, can be properly verified.

We should also consider that in Linux-like data mode only user code is verified. The code included into the system include-files on Windows will anyway remain such as in the case of LLP64 memory model. As we know, all the files are processed by a preprocessor before the analysis, which of course knows nothing about LP64 model.

In order to better demonstrate the peculiarities of PVS-Studio use in "Linux-like mode", here are some examples cited:

```
void foo(ptrdiff_t delta);
int i = -2;
unsigned k = 1;
foo(i + k);
```

The given code will be potentially dangerous both for Windows and Linux systems. The incorrect result happens during implicit actual argument extension, which has a meaning of 0xFFFFFFFF unsigned type, up to ptrdiff_t type. In this case, the warning will be given regardless of the mode used.

```
void *ptr;
long np = (long) (ptr);
```

The given code diagnosed as incorrect in 64-bit Windows programs will be considered correct for 64-bit Linux systems.

The following example, on the contrary, demonstrates a code which is correct for Windows systems but is incorrect for Linux:

```
long array[4] = { 1, 2, 3, 4 };
enum ENumbers { ZERO, ONE, TWO, THREE, FOUR };
int *intPtr = (int *) (array);
cout << intPtr[1] << endl;
```

As the sizes of int and long in 64-bit Linux systems do not correspond, the analyzer will give a respective diagnostic message.

The last two examples demonstrate the restrictions of the new functioning mode:

```
#ifdef _WINDOWS
    typedef long MY_SIGNED_32;
#else //Linux
    typedef int MY_SIGNED_32;
#endif
```

In such situations, as the preprocessor will select the branch "typedef long MY_SIGNED_32;" all the code constructions in which MY_SIGNED_32 type will be used will be incorrectly diagnosed from the point of view of Linux systems.

Even a simpler example can be cited in which the code that is to be analyzed will not be analyzed:

```
#ifdef LINUX_64
    // Much code here
#endif
```

Despite the listed restrictions, the possibility of work both with the traditional data model for Windows LLP64 and the traditional for Linux model LP64 widens the code analyzer range of application. It is, however, important to understand the way this option works, what it can provide and what it cannot.

By default, the new mode is turned off, and users for whom work in Linux-environment is not of relevance, can ignore it.

TypesForNonLargeArrays

When detecting [64-bit errors](#) with the Viva64 rule set, the analyzer generates many false alarms. Unfortunately, you cannot avoid it due to specifics of this diagnostic type. However, we try to employ various methods to make false alarms fewer. One of these methods is introduction of a notion of a type that does not create large arrays.

The analyzer has to generate the [V108](#) warning on the code in the following function because it does not know if the array can contain more than INT_MAX (2^31) items of the MY_STRUCT type.

```
struct MY_STRUCT { int a; };
class MyArray {
    MY_STRUCT *m_arr;
    MY_STRUCT &Get(int i) {
        return m_arr[i]; //V108
    }
};
```

If there cannot be many items of the MY_STRUCT type, the code is safe. The 'int' type is enough to access any item of the 'm_arr' array. You can tell the PVS-Studio analyzer what types will not be used to create many items. To do that, you can use the "Custom" option. You may specify there the file name that will contain the list of types. The file must be in the text format. Each line in the file contains a type name. For instance, it might look this way:

```
MyType_A
MyHandle
MyPoint
```

The analyzer by default offers you a set of types that must not be used in arrays of large size. For example, here are some of these types:

WinAPI: HWND, HMENU, HMODULE, HCURSOR, ...

MFC/ATL: CWnd, CComPtr, CTabCtrl, CDC, ...

WX widgets: wxString, wxBrush, wxDC, wxPrintData, ...

QT: QPalette, QFontMetrics, QLocale, QCursor, ...

STL: deque, string, vector, set, ...

An array consisting of more than two billions of items of the CWnd type is just senseless. You simply cannot create so many windows.

It is not so evident with other types such as deque. I stress that we speak not of the size of the deque container – we mean the size of an array consisting of deque-items. If the size of such an array approaches INT_MAX, using deque is not a very good architectural solution and you should consider creating your own data structure for the task being solved.

That is why the analyzer by default considers that the program does not have arrays containing more than INT_MAX (2^31) strings of the std::string type and so on. If it is not so, the user may disable the function when the analyzer considers some arrays safe. To do this, you may disable in the settings some type sets for the popular libraries: MFC, Qt, STL, WinAPI, wxWidgets.

Note. If you have noticed that the analyzer generates a false alarm where you are sure that there cannot be many objects of some type, write to our support service. Specify the type name and library where you have encountered this type. We will add it into exceptions in the next version.

V001. A code fragment from 'file' cannot be analyzed.

The analyzer sometimes fails to diagnose a file with source code completely. There may be three reasons for that:

1) An error in code

There is a template class or template function with an error. If this function is not instantiated, the compiler fails to detect some errors in it. In other words, such an error does not hamper compilation. The PVS-Studio tries to find potential errors even in classes and functions that are not used anywhere. If the analyzer cannot parse some code, it will generate the V001 warning. Consider a code sample:

```
template <class T>
class A
{
public:
    void Foo()
    {
        //forget ;
        int x
    }
};
```

Visual C++ will compile this code if the A class is not used anywhere. But it contains an error, which hampers PVS-Studio's work.

2) An error in the Visual C++'s preprocessor

The analyzer uses the Visual C++'s preprocessor while working. From time to time this preprocessor makes errors when generating preprocessed "*.i" files. As a result, the analyzer receives incorrect data. Here is a sample:

```
hWnd = CreateWindow (
    wndclass.lpszClassName,    // window class name
    T("NcFTPBatch"),          // window caption
    WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX,
    // window style
    100,                      // initial x position
    100,                      // initial y position
    450,                      // initial x size
    100,                      // initial y size
    NULL,                    // parent window handle
    NULL,                    // window menu handle
    hInstance,               // program instance handle
    NULL);                   // creation parameters
if (hWnd == NULL) {
    ...
```

Visual C++'s preprocessor turned this code fragment into:

```
hWnd = // window class name// window caption// window style//
initial x position// initial y position// initial x size//
initial y size// parent window handle// window menu handle//
program instance handleCreateWindowExA(0L,
wndclass.lpszClassName, "NcFTPBatch", 0x00000000L | 0x00C00000L |
0x00080000L | 0x00020000L, 100, 100, 450, 100, ((void *)0),
((void *)0), hInstance, ((void *)0)); // creation parameters
if (hWnd == NULL) {
    ...
```

It turns out that we have the following code:

```
hWnd = // a long comment
if (hWnd == NULL) {
    ...
}
```

This code is incorrect and PVS-Studio will inform you about it. Of course it is a defect of PVS-Studio, so we will eliminate it in time.

I want also to note that Visual C++ successfully compiles this code because the algorithms it uses for compilation purposes and generation of preprocessed "*.i" files are different.

3) Defects inside PVS-Studio

On rare occasions PVS-Studio fails to parse complex template code.

Whatever the reason for generating the V001 warning, it is not crucial. Usually incomplete parse of a file is not very significant from the viewpoint of analysis. PVS-Studio simply skips a function/class with an error and continues analyzing the file. Only a small code fragment is left unanalyzed.

V002. Some diagnostic messages may contain incorrect line number.

The analyzer can sometimes output the error "Some diagnostic messages may contain incorrect line number". This occurs in two cases:

1. Use of multiline `#define`-macros on Microsoft Visual Studio 2005 system without Service Pack 1.
2. Use of multiline `#pragma` directives on all supported versions of Microsoft Visual Studio.

Any code analyzer works only with preprocessed files, i.e. with those files in which all macros (`#define`) are substituted, and all include files are substituted (`#include`). With this, in the pre-processed file, information is contained about which files were substituted in which positions. That means, in the preprocessed files, information about line numbers is contained.

Preprocessing is carried out in any case. For the user, this procedure looks quite transparent. Sometimes, the preprocessor is a part of the code analyzer, and sometimes (like in the case with PVS-Studio) external preprocessor is used. In PVS-Studio, preprocessor by Microsoft Visual Studio is used. The analyzer starts the command line compiler `cl.exe` for each C/C++ file being processed and with its help, generates preprocessed file with ".i" extension.

There exists an error in the preprocessor by Microsoft Visual Studio 2005. If preprocessing is done from the command line (the way it is done in PVS-Studio), then in case multiline macros are present in the program, information about line numbers becomes confused. This can lead to incorrect positioning of the code analyzer in the file. It means that the code analyzer will find the real problem, but it will show incorrectly the line in which this error was detected.

Let us explain it on an example. In a simple code, `assert` is used, with this, the expression in parentheses is divided into several lines. E.g., this way:

```
int _tmain(int argc, _TCHAR* argv[])
{
    int a = 0;
    int b = 1;
    size_t c = 2;
    assert(a ==
        b);
    a++; // the analyzer will show an error in this harmless line.
    c = a; // actually, however, this diagnostic message should
           // point at the line "c = a;".
    return 0;
}
```

As we can see it from the remarks, because multiline `assert` is present, the analyzer will show that there exists an error (this is correct), but as the line that contains this error the above one will be shown. Of course, this can mislead the user.

If the same code is written with `assert` in one line, there will be no problem:

```
int _tmain(int argc, _TCHAR* argv[])
{
    int a = 0;
    int b = 1;
    size_t c = 2;
    assert(a == b);
    a++;
    c = a; // Diagnostic message refers to
           // the correct line.
    return 0;
}
```

This error shows itself only during PVS-Studio work on Microsoft Visual Studio 2005 without Visual Studio Service Pack 1. On Visual Studio 2005 Service Pack1, as well as on higher program versions (Visual Studio 2008 and higher) this problem does not exist. We recommend the following solution of the problem: install Visual Studio 2005 Service Pack 1 on Visual Studio 2005. Then the code with multiline macros will be processed by PVS-Studio analyzer correctly.

Here is one more situation at which the message "Some diagnostic messages may contain incorrect line number" is output, and a failure in positioning diagnostic messages occurs. It is referred to multiline directives `#pragma` of a special type. Here is an example of correct code:

```
#pragma warning(push)
void test()
{
    int a = 0;
    size_t b = a; // PVS-Studio will inform about the error here
}
#pragma warning(pop)
```

If `#pragma` directive is written in two lines, the analyzer PVS-Studio will show an error in an incorrect place (there will be a one-line shift):

```
#pragma \
warning(push)
void test()
{
    int a = 0; // PVS-Studio will show the error here,
    size_t b = a; // actually, however, the error should be here.
}
#pragma warning(pop)
```

In another case, however, with multiline `#pragma` directive there will be no error:

```
#pragma warning \
(push)
void test()
{
    int a = 0;
    size_t b = a; // PVS-Studio will inform about the error in this line
}
#pragma warning(pop)
```

This error is corrected by installing Service Pack 1 on Visual Studio 2005 or upgrade to Visual Studio 2008. The only recommendation is either not to use

multiline `#pragma` directives or to use them, but in the way they are correctly processed.

The code analyzer tries to detect a failure in lines numbering in the processed file. This mechanism is an heuristic one, and it cannot guarantee correct determination of positioning diagnostic messages in the program code. However, if it is possible to realize that a particular file contains multiline macros, and there exists a positioning error, then the message "Some diagnostic messages may contain incorrect line number" is given out.

This mechanism operates the following way.

The analyzer opens the source C/C++ file and searches for the very last token. Tokens not shorter than three symbols are selected only in order to ignore closing parentheses, etc. E.g., for the following code, operator "return" will be considered the last token:

```
01 #include "stdafx.h"
02
03 int foo(int a)
04 {
05     assert(a >= 0 &&
06           a <= 1000);
07     int b = a + 1;
08     return b;
09 }
```

Having found the last token, the analyzer will determine the number of line which contains it. In this very case, it is line 8. Further on, the analyzer searches for the last token in the file which has already been preprocessed. If the last tokens do not coincide, the macro in the end of file has been likely to substitute; the analyzer is unable to understand whether the lines are arranged correctly, and ignores the given situation. However, such situations occur very rarely, and last tokens almost always coincide in the source and preprocessed files. If it is so, the number of line, in which the token in the preprocessed file is situated, is determined.

Thus, we have the line numbers in which the last token is situated in the source file and in the preprocessed file. If these line numbers do not coincide, then there has been a failure in lines numbering. In this case, the analyzer will notify the user about it with the message "Some diagnostic messages may contain incorrect line number".

Please consider that if a multiline macro or a multiline `#pragma`-directive are situated in the file after all the dangerous code areas found, then all the line numbers for the found errors will be correct. Even though the analyzer outputs the message "Some diagnostic messages may contain incorrect line number for file", this will not prevent you from analyzing the diagnostic messages given out by it.

Please pay attention that though this is not an error purely of PVS-Studio code analyzer, it anyway leads to incorrect work of the code analyzer.

V003. Unrecognized error found...

The message V003 means that a critical error has occurred in the analyzer. It is most likely that in this case you will not see any warning messages concerning the file being checked at all.

Although the message V003 is very rare, we will appreciate if you help us fix the issue that has caused this message to appear. To do this, please send us a preprocessed i-file that caused the error by this e-mail support@viva64.com.

Note. A preprocessed i-file results from a source file (for example, file.cpp) when the preprocessor finishes its work. To get this file you should set the option `RemoveIntermediateFiles` to `False` on the tab "Common Analyzer Settings" in PVS-Studio settings and relaunch analysis of this one file. After that you can find the corresponding i-file in the project folder (for example, file.i).

V004. For a more precise verification, please select x64 or Itanium configuration instead of Win32.

When detecting issues of 64-bit code, it is 64-bit configuration of a project that the analyzer must always test. For it is 64-bit configuration where data types are correctly expanded and branches like `"#ifdef WIN64"` are selected, and so on. It is incorrect to try to detect issues of 64-bit code in a 32-bit configuration.

But sometimes it may be helpful to test the 32-bit configuration of a project. You can do it in case when there is no 64-bit configuration yet but you need to estimate the scope of work on porting the code to a 64-bit platform. In this case you can test a project in 32-bit mode. Testing the 32-bit configuration instead of the 64-bit one will show how many diagnostic warnings the analyzer will generate when testing the 64-bit configuration. Our experiments show that of course far not all the diagnostic warnings are generated when testing the 32-bit configuration. But about 95% of them in the 32-bit mode coincide with those in the 64-bit mode. It allows you to estimate the necessary scope of work.

Pay attention! Even if you correct all the errors detected when testing the 32-bit configuration of a project, you cannot consider the code fully compatible with 64-bit systems. You need to perform the final testing of the project in its 64-bit configuration.

The V004 message is generated only once for each project checked in the 32-bit configuration. The warning refers to the file which will be the first to be analyzed when checking the project. It is done for the purpose to avoid displaying a lot of similar warnings in the report.

V005. Cannot determine active configuration for project. Please check projects and solution configurations.

This issue with PVS-Studio is caused by the mismatch of selected project's platform configurations declared in the solution file (Vault.sln) and platform configurations declared in the project file itself. For example, the solution file may contain the lines of this particular kind for concerned project:

```
{F56ECFEC-45F9-4485-8A1B-6269E0D27E49}.Release|x64.ActiveCfg = Release|x64
```

However, the project file itself may lack the declaration of `Release|x64` configuration. Therefore while trying to check this particular project PVS-Studio is unable to locate the 'Release|x64' configuration. The following line is expected to be automatically generated by IDE in the solution file for such a case:

```
{F56ECFEC-45F9-4485-8A1B-6269E0D27E49}.Release|x64.ActiveCfg = Release|Win32
```

In automatically generated solution file the solution's active platform configuration (`Release|x64.ActiveCfg`) is set equal to one of project's existing configurations (i.e. in this particular case `Release|Win32`). Such a situation is expected by and can be handled by PVS-Studio correctly.

V006. File cannot be processed. Analysis aborted by timeout.

Message V006 is generated when an analyzer's process cannot process some file for a particular time period and aborts. This might happen in two cases.

The first reason is an error inside the analyzer that does not allow it to parse some code fragment. It happens rather seldom yet it is possible. Although message V006 appears rather seldom, we would appreciate if you help us to eliminate the issue which causes the message to appear. Please send your preprocessed i-file at which this issue occurs to the address support@viva64.com.

Note. A preprocessed i-file is generated from the source file (for instance, file.cpp) after the preprocessor has analyzed the file. To get this file, you should go

to the "Common Analyzer Settings" tab in the PVS-Studio's settings and set the option `RemoveIntermediateFiles` into `False` and perform analysis of this single file once again. After that, you can find the corresponding `i`-file in the project's folder (for instance, `file.i`).

The second possible reason is the following: although the analyzer could process the file correctly, it does not have enough time to do that because it gets too few system resources due to high processor load. By default, the number of threads spawned for analysis is equal to the number of processor cores. For example, if we have four cores in our machine, the tool will start analysis of four files at once. Each instance of an analyzer's process requires about 1.5 Gbytes of memory. If your computer does not have enough memory, the tool will start using the swap file and analysis will run slowly and fail to fit into the required time period. Besides, you may encounter this problem when you have other "heavy" applications running on your computer simultaneously with the analyzer.

To solve this issue, you may directly restrict the number of cores to be used for analysis in the PVS-Studio's settings (`ThreadCount` option on the "Common Analyzer Settings" tab).

V007. Verification of CLR projects is not implemented. Incorrect diagnostics are possible.

The V007 message appears then the projects utilizing the C++/Common Language Infrastructure Microsoft specification (which itself supersedes Microsoft's Managed Extensions for C++) are selected for analysis. The C++/CLI specification, as opposed to Managed Extensions for C++, can be viewed as a separate language independent from C++ with its own syntax and keywords. Because of this PVS-Studio analyzer can miss some of the errors present in the files which contain CLI specific syntax. Although if some individual files do not contain such syntax and it is possible to verify and find errors in them, PVS-Studio officially does not support such a behavior and the verification of all files included in C++/CLI projects is not guaranteed.

V008. Unable to start the analysis on this file.

V008. Unable to start the analysis on this file.

PVS-Studio was unable to start the analysis of the designated file. This message indicates that an external C++ preprocessor, started by the analyzer to create preprocessed source code file, had exited with a non-zero error code. Moreover, `std` error could also contain the detailed description of this error, which can be reviewed for the file in question with PVS-Studio Output Window.

There could be several reasons for the V008 error:

1) The source code is not compilable

If the C++ sources code is not compilable for some reason (a missing header file for example), then the preprocessor will exit with non-zero error code and the "fatal compilation error" type message will be outputted into `std` error. PVS-Studio is unable to initiate the analysis in case C++ file hadn't been successfully preprocessed. To resolve this error you should ensure the compilability of the file being analyzed.

2.) The preprocessor's executable file had been damaged/locked

Such a situation is possible when the preprocessor's executable file had been damaged or locked by system antiviral software. In this case the PVS-Studio Output window could also contain the error messages of this kind: "The system cannot execute the specified program". To resolve it you should verify the integrity of the utilized preprocessor's executable and lower the security policies' level of system's antiviral software.

3.) One of PVS-Studio auxiliary command files had been locked.

PVS-Studio analyzer is not launching the C++ preprocessor directly, but with the help of its own pre-generated command files. In case of strict system security policies, antiviral software could potentially block the correct initialization of C++ preprocessor. This could be also resolved by easing the system security policies toward the analyzer.

V009. Intel C++ project toolset is not supported. Select Visual C++ toolset to verify this project.

V009. Intel C++ project toolset is not supported. Select Visual C++ toolset to verify this project.

This error message indicates that PVS-Studio had detected the presence of Intel C++ compiler integration within specified Visual C++ project's build configuration. PVS-Studio analyzer supports the verification of only the Microsoft Visual C++ (`cl.exe`) oriented projects.

Although PVS-Studio is unable to verify build configurations which are utilizing Intel C++ compiler, it is still possible to verify the project in question by modifying the *platform toolset* field in the project's properties (Properties>General>Platform Toolset) in accordance with platform toolsets supported by the analyzer. At this moment PVS-Studio supports the following platform toolsets:

- Microsoft Visual C++ 8.0 compiler (v80, installed with MSVS 2005)
- Microsoft Visual C++ 9.0 compiler (v90, installed with MSVS 2008)
- Microsoft Visual C++ 10.0 compiler (v100, default value for MSVS 2010)

The V009 message concerns only Visual Studio 2010 projects (`vcxproj` extension), as the earlier Visual C++ project format (`vcproj`) does not support the use of third-party (i.e. different from Visual C++ 10.0) compilers through direct native platform toolset properties.

In the earlier versions of Microsoft Visual Studio (2005 and 2008), Intel C++ compiler is integrated into the IDE's build process by the addition of its own projects of a distinctive type (different from the standard `vcproj`) into the active solution. PVS-Studio analyzer does not support the verification of such modified Intel C++ project types.

V101. Implicit assignment type conversion to memsize type.

The analyzer detected a potential error relating to [implicit type conversion](#) while executing the assignment operator `=`. The error may consist in incorrect calculating of the value of the expression to the right of the assignment operator `=`. An example of the code causing the warning message:

```
size_t a;
unsigned b;
...
a = b; // V101
```

The operation of converting a 32-bit type to [memsize](#)-type is safe in itself as there is no data loss. For example, you can always save the value of unsigned-type variable into a variable of `size_t` type. But the presence of this type conversion may indicate a hidden error made before.

The first cause of the error occurrence on a 64-bit system may be the change of the expression calculation process. Let's consider an example:

```
unsigned a = 10;
int b = -11;
ptrdiff_t c = a + b; //V101
cout << c << endl;
```

On a 32-bit system this code will display the value -1, while on a 64-bit system it will be 4294967295. This behaviour fully meets the rules of type conversion in C++ but most likely it will cause an error in a real code.

Let's explain the example. According to C++ rules `a+b` expression has unsigned type and contains the value `0xFFFFFFFFu`. On a 32-bit system `ptrdiff_t` type is a sign 32-bit type. After `0xFFFFFFFFu` value is assigned to the 32-bit sign variable it will contain the value -1. On a 64-bit system `ptrdiff_t` type is a sign 64-bit type. It means `0xFFFFFFFFu` value will be represented as it is. That is, the value of the variable after assignment will be 4294967295.

The error may be corrected by excluding mixed use of `memsize` and `non-memsize`-types in one expression. An example of code correction:

```
size_t a = 10;
ptrdiff_t b = -11;
ptrdiff_t c = a + b;
cout << c << endl;
```

A more proper way of correction is to refuse using sign and non-sign data types together.

The second cause of the error may be an overflow occurring in 32-bit data types. In this case the error may stand before the assignment operator but you can detect it only indirectly. Such errors occur in code allocating large memory sizes. Let's consider an example:

```
unsigned Width  = 1800;
unsigned Height = 1800;
unsigned Depth  = 1800;
// Real error is here
unsigned CellCount = Width * Height * Depth;
// Here we get a diagnostic message V101
size_t ArraySize = CellCount * sizeof(char);
cout << ArraySize << endl;
void *Array = malloc(ArraySize);
```

Suppose that we decided to process data arrays of more than 4 Gb on a 64-bit system. In this case the given code will cause allocation of a wrong memory size. The programmer is planning to allocate 5832000000 memory bytes but he gets only 1537032704 instead. It happens because of an overflow occurring while calculating `Width * Height * Depth` expression. Unfortunately, we cannot diagnose the error in the line containing this expression but we can indirectly indicate the presence of the error detecting type conversion in the line:

```
size_t ArraySize = CellCount * sizeof(char); //V101
```

To correct the error you should use types allowing you to store the necessary range of values. Mind that correction of the following kind is not appropriate:

```
size_t CellCount = Width * Height * Depth;
```

We still have the overflow here. Let's consider two examples of proper code correction:

```
// 1)
unsigned Width  = 1800;
unsigned Height = 1800;
unsigned Depth  = 1800;
unsigned CellCount =
    static_cast<size_t>(Width) *
    static_cast<size_t>(Height) *
    static_cast<size_t>(Depth);
// 2)
size_t Width  = 1800;
size_t Height = 1800;
size_t Depth  = 1800;
size_t CellCount = Width * Height * Depth;
```

You should keep in mind that the error can be situated not only higher but even in another module. Let's give a corresponding example. Here the error consists in incorrect index calculation when the array's size exceeds 4 Gb.

Suppose that the application uses a large one-dimensional array and `CalcIndex` function allows you to address this array as a two-dimensional one.

```
extern unsigned ArrayWidth;
unsigned CalcIndex(unsigned x, unsigned y) {
    return x + y * ArrayWidth;
}
...
const size_t index = CalcIndex(x, y); //V101
```

The analyzer will warn about the problem in the line: `const size_t index = CalcIndex(x, y)`. But the error is in incorrect implementation of `CalcIndex` function. If we take `CalcIndex` separately it is absolutely correct. The output and input values have unsigned type. Calculations are also performed only with unsigned types participating. There are no explicit or implicit type conversions and the analyzer has no opportunity to detect a logic problem relating to `CalcIndex` function. The error consists in that the result returned by the function and possibly the result of the input values was chosen incorrectly. The function's result must have `memsize` type.

Fortunately, the analyzer managed to detect implicit conversion of `CalcIndex` function's result to `size_t` type. It allows you to analyze the situation and bring necessary changes into the program. Correction of the error may be, for example, the following:

```
extern size_t ArrayWidth;
size_t CalcIndex(size_t x, size_t y) {
    return x + y * ArrayWidth;
}
...
const size_t index = CalcIndex(x, y);
```

If you are sure that the code is correct and the array's size will never reach 4 Gb you can suppress the analyzer's warning message by [explicit type conversion](#):

```
extern unsigned ArrayWidth;
unsigned CalcIndex(unsigned x, unsigned y) {
    return x + y * ArrayWidth;
}
...
const size_t index = static_cast<size_t>(CalcIndex(x, y));
```

In some cases the analyzer can understand itself that an overflow is impossible and the message won't be displayed.

Let's consider the last example related to incorrect shift operations

```
ptrdiff_t SetBitN(ptrdiff_t value, unsigned bitNum) {
    ptrdiff_t mask = 1 << bitNum; //V101
    return value | mask;
}
```

The expression `"mask = 1 << bitNum"` is unsafe because this code cannot set the high-order bits of the 64-bit variable `mask` into ones. If you try to use `SetBitN` function for setting, for example, the 33rd bit, an overflow will occur when performing the shift operation and you will not get the result you've expected.

V102. Usage of non memsize type for pointer arithmetic.

The analyzer found a possible error in pointer arithmetic. The error may be caused by an overflow during the determination of the expression.

Let's take up the first example.

```
short a16, b16, c16;
char *pointer;
...
pointer += a16 * b16 * c16;
```

The given example works correctly with pointers if the value of the expression $a16 * b16 * c16$ does not exceed *INT_MAX* (2Gb). This code could always work correctly on the 32-bit platform because the program never allocated large-sized arrays. On the 64-bit platform the programmer using the previous code while working with an array of a large size would be disappointed. Suppose, we would like to shift the pointer value in 3000000000 bytes, and the variables *a16*, *b16* and *c16* have values 3000, 1000 and 1000 correspondingly. During the determination of the expression $a16 * b16 * c16$ all the variables, according to the C++ rules, will be converted to type *int*, and only then the multiplication will take place. While multiplying an overflow will occur, and the result of this would be the number -1294967296. The incorrect expression result will be extended to type *ptrdiff_t* and pointer determination will be launched. As a result, we'll face an abnormal program termination while trying to use the incorrect pointer.

To prevent such errors one should use [memsize](#) types. In our case it will be correct to change the types of the variables *a16*, *b16*, *c16* or to use the [explicit type conversion](#) to type *ptrdiff_t* as follows:

```
short a16, b16, c16;
char *pointer;
...
pointer += static_cast<ptrdiff_t>(a16) *
           static_cast<ptrdiff_t>(b16) *
           static_cast<ptrdiff_t>(c16)
```

It's worth mentioning that it is not always incorrect not to use memsize type in pointer arithmetic. Let's examine the following situation:

```
char ch;
short a16;
int *pointer;
...
int *decodePtr = pointer + ch * a16;
```

The analyzer does not show a message on it because it is correct. There are no determinations which may cause an overflow and the result of this expression will be always correct on the 32-bit platform as well as on the 64-bit platform.

V103. Implicit type conversion from memsize type to 32-bit type.

The analyzer found a possible error related to the implicit [memsize](#)-type conversion to 32-bit type. The error consists in the loss of high bits in 64-bit type which causes the loss of the value.

The compiler also diagnoses such type conversions and shows warnings. Unfortunately, such warnings are often switched off, especially when the project contains a great deal of the previous legacy code or old libraries are used. In order not to make a programmer look through hundreds and thousands of such warnings, showed by the compiler, the analyzer informs only about those which may be the cause of the incorrect work of the code on the 64-bit platform.

The first example.

Our application works with videos and we want to calculate what file-size we'll need in order to store all the shots kept in memory into a file.

```
size_t Width, Height, FrameCount;
...
unsigned BufferSizeForWrite = Width * Height * FrameCount * sizeof(RGBStruct);
```

Earlier the general size of the shots in memory could never exceed 4 Gb (practically 2-3 Gb depending on the kind of OS Windows). On the 64-bit platform we have an opportunity to store much more shots in memory, and let's suppose that their general size is 10 Gb. After putting the result of the expression $Width * Height * FrameCount * sizeof(RGBStruct)$ into the variable *BufferSizeForWrite*, we'll truncate high bits and will deal with the incorrect value.

The correct solution will be to change the type of the variable *BufferSizeForWrite* into type *size_t*.

```
size_t Width, Height, FrameCount;
...
size_t BufferSizeForWrite = Width * Height * FrameCount * sizeof(RGBStruct);
```

The second example.

Saving of the result of pointers subtraction.

```
char *ptr_1, *ptr_2;
...
int diff = ptr_2 - ptr_1;
```

If pointers differ more than in one *INT_MAX* byte (2 Gb) a value cutoff during the assignment will occur. As a result the variable *diff* will have an incorrect value. For the storing of the given value we should use type *ptrdiff_t* or another memsize type.

```
char *ptr_1, *ptr_2;
...
ptrdiff_t diff = ptr_2 - ptr_1;
```

When you are sure about the correctness of the code and the implicit type conversion does not cause errors while changing over to the 64-bit platform, you may use the [explicit type conversion](#) in order to avoid error messages showed in this line. For example:

```
unsigned BitCount = static_cast<unsigned>(sizeof(RGBStruct) * 8);
```

If you suspect that the code contains incorrect explicit conversions of memsize types to 32-bit types about which the analyzer does not warn, you can use the [V202](#).

As was said before analyzer informs only about those type conversions which can cause incorrect code work on a 64-bit platform. The code given below won't be considered incorrect though there occurs conversion of memsize type to int type:

```
int size = sizeof(float);
```

V104. Implicit type conversion to memsize type in an arithmetic expression.

The analyzer found a possible error inside an arithmetic expression and this error is related to the [implicit type conversion](#) to [memsize](#) type. The error of an overflow may be caused by the changing of the permissible interval of the values of the variables included into the expression. The first example.

The incorrect comparison expression. Let's examine the code:

```
size_t n;
unsigned i;
// Infinite loop (n > UINT_MAX).
for (i = 0; i != n; ++i) { ... }
```

In this example the error are shown which are related to the implicit conversion of type *unsigned* to type *size_t* while performing the comparison operation.

On the 64-bit platform you may have an opportunity to process a larger data size and the value of the variable *n* may exceed the number *UINT_MAX* (4 Gb). As a result, the condition *i != n* will be always true and that will cause an eternal cycle.

An example of the corrected code:

```
size_t n;
size_t i;
for (i = 0; i != n; ++i) { ... }
```

The second example.

```
char *begin, *end;
int bufLen, bufCount;
...
ptrdiff_t diff = begin - end + bufLen * bufCount;
```

The implicit conversion of type *int* to type *ptrdiff_t* often indicates an error. One should pay attention that the conversion takes place not while performing operator "=" (for the expression *begin - end + bufLen * bufCount* has type *ptrdiff_t*), but inside this expression. The subexpression *begin - end* according to C++ rules has type *ptrdiff_t*, and the right *bufLen * bufCount* type *int*. While changing over to 64-bit platform the program may begin to process a larger data size which may result in an overflow while determining the subexpression *bufLen * bufCount*.

You should change the type of the variables *bufLen* and *bufCount* into memsize type or use the [explicit type conversion](#), as follows:

```
char *begin, *end;
int bufLen, bufCount;
...
ptrdiff_t diff = begin - end + ptrdiff_t(bufLen) * ptrdiff_t(bufCount);
```

Let's notice that the implicit conversion to memsize type inside the expressions is not always incorrect. Let's examine the following situation:

```
size_t value;
char c1, c2;
size_t result = value + c1 * c2;
```

The analyzer does not show error message although the conversion of type *int* to *size_t* occurs in this case, for there can be no overflow while determining the subexpression *c1 * c2*.

If you suspect that the program may contain errors related to the incorrect explicit type conversion in expressions, you may use the [V201](#). Here is an example when the explicit type conversion to type *size_t* hides an error:

```
int i;
size_t st;
...
st = size_t(i * i * i) * st;
```

V105. N operand of '?' operation: implicit type conversion to memsize type.

The analyzer found a possible error inside an arithmetic expression related to the [implicit type conversion](#) to [memsize](#) type. An overflow error may be caused by the changing of the permissible interval of the values of the variables included into the expression. This warning is almost equivalent to warning [V104](#) with the exception that the implicit type conversion occurs due to the use of ? : operation.

Let's give an example of the implicit type conversion while using operation:

```
int i32;
float f = b != 1 ? sizeof(int) : i32;
```

In the arithmetic expression the ternary operation ? : is used which has three operands:

- *b != 1* - the first operand;
- *sizeof(int)* - the second operand;
- *i32* - the third operand.

The result of the expression *b != 1 ? sizeof(int) : i32* is the value of type *size_t* which is then converted into type *float* value. Thus, the implicit type conversion realized for the 3rd operand of ? : operation.

Let's examine an example of the incorrect code:

```
bool useDefaultVolume;
size_t defaultVolume;
unsigned width, height, depth;
...
size_t volume = useDefaultVolume ?
                defaultVolume :
                width * height * depth;
```

Let's suppose, we're developing an application of computational modeling which requires three-dimensional calculation area. The number of calculating elements which are used is determined according to the variable *useDefaultSize* value and is assigned on default or by multiplication of length, height and depth of the calculating area. On the 32-bit platform the size of memory which was already allocated, cannot exceed 2-3 Gb (depending on the kind of OS Windows) and as consequence the result of the expression *width * height * depth* will be always correct. On the 64-bit platform, using the opportunity to deal with a larger memory size, the number of calculating elements may exceed the value *UINT_MAX* (4 Gb). In this case an overflow will occur while determining the expression *width * height * depth* because the result of this expression had type *unsigned*.

Correction of the code may consist in the changing of the type of the variables *width*, *height* and *depth* to memsize type as follows:

```
...
size_t width, height, depth;
```

```
...
size_t volume = useDefaultVolume ?
                defaultVolume :
                width * height * depth;
```

Or in use of the [explicit type conversion](#):

```
unsigned width, height, depth;
...
size_t volume = useDefaultVolume ?
                defaultVolume :
                size_t(width) * size_t(height) * size_t(depth);
```

In addition, we advise to read the description of a similar warning [V104](#), where one can learn about other effects of the implicit type conversion to memsize type.

V106. Implicit type conversion N argument of function 'foo' to memsize type.

The analyzer found a possible error related to the implicit actual function argument conversion to [memsize](#) type.

The first example.

The program deals with large arrays using container *CArray* from library MFC. On the 64-bit platform the number of array items may exceed the value *MAX_INT* (2Gb), which will make the work of the following code impossible:

```
CArray<int, int> myArray;
...
int invalidIndex = 0;
INT_PTR validIndex = 0;
while (validIndex != myArray.GetSize()) {
    myArray.SetAt(invalidIndex, 123);
    ++invalidIndex;
    ++validIndex;
}
```

The given code fills all the array *myArray* items with value 123. It seems to be absolutely correct and the compiler won't show any warnings in spite of its impossibility to work on the 64-bit platform. The error consists in the use of type *int* as an index of the variable *invalidIndex*. When the value of the variable *invalidIndex* exceeds *MAX_INT* an overflow will occur and it will receive value "-1". The analyzer diagnoses this error and warns that the [implicit conversion](#) of the first argument of the function *SetAt* to memsize type (here it is type *INT_PTR*) occurs. When seeing such a warning you may correct the error replacing *int* type with a more appropriate one.

The given example is significant because it is rather unfair to blame a programmer for the ineffective code. The reason is that *GetAt* function in class *CArray* in the previous MFC library version was declared as follows:

```
void SetAt(int nIndex, ARG_TYPE newElement);
```

And in the new version:

```
void SetAt(INT_PTR nIndex, ARG_TYPE newElement);
```

Even the Microsoft developers creating MFC could not take into account all the possible consequences of the use of *int* type for indexing in the array and we can forgive the common developer who has written this code.

Here is the correct variant:

```
...
INT_PTR invalidIndex = 0;
INT_PTR validIndex = 0;
while (validIndex != myArray.GetSize()) {
    myArray.SetAt(invalidIndex, 123);
    ++invalidIndex;
    ++validIndex;
}
```

The second example.

The program determines the necessary data array size and then allocated it using function *malloc* as follows:

```
unsigned GetArraySize();
...
unsigned size = GetArraySize();
void *p = malloc(size);
```

The analyzer will warn about the line *void *p = malloc(size);*. Looking through the definition of function *malloc* we will see that its formal argument assigning the size of the allocated memory is represented by type *size_t*. But in the program the variable *size* of *unsigned* type is used as the actual argument. If your program on the 64-bit platform needs an array more than *UINT_MAX* bytes (4Gb), we can be sure that the given code is incorrect for type *unsigned* cannot keep a value more than *UINT_MAX*. The program correction consists in changing the types of the variables and functions used in the determination of the data array size. In the given example we should replace *unsigned* type with one of memsize types, and also if it is necessary modify the function *GetArraySize* code.

```
...
size_t GetArraySize();
...
size_t size = GetArraySize();
void *p = malloc(size);
```

The analyzer show warnings on the implicit type conversion only if it may cause an error during program port on the 64-bit platform. Here it is the code which contains the implicit type conversion but does not cause errors:

```
void MyFoo(SSIZE_T index);
...
char c = 'z';
MyFoo(0);
MyFoo(c);
```

If you are sure that the implicit type conversion of the actual function argument is absolutely correct you may use the [explicit type conversion](#) to suppress the analyzer's warnings as follows:

```
typedef size_t TYear;
void MyFoo(TYear year);
int year;
...
MyFoo(static_cast<TYear>(year));
```

Sometimes the explicit type conversion may hide an error. In this case you may use the [V201](#).

V107. Implicit type conversion N argument of function 'foo' to 32-bit type.

The analyzer found a possible error related to the [implicit conversion](#) of the actual function argument which has [memsize](#) type to 32-bit type.

Let's examine an example of the code which contains the function for searching for the max array item:

```
float FindMaxItem(float *array, int arraySize) {
    float max = -FLT_MAX;
    for (int i = 0; i != arraySize; ++i) {
        float item = *array++;
        if (max < item)
            max = item;
    }
    return max;
}
...
float *beginArray;
float *endArray;
float maxValue = FindMaxItem(beginArray, endArray - beginArray);
```

This code may work successfully on the 32-bit platform but it won't be able to process arrays containing more than *INT_MAX* (2Gb) items on the 64-bit architecture. This limitation is caused by the use of *int* type for the argument *arraySize*. Pay attention that the function code looks absolutely correct not only from the compiler's point of view but also from that of the analyzer. There is no type conversion in this function and one cannot find the possible problem.

The analyzer will warn about the implicit conversion of memsize type to a 32-bit type during the invocation of *FindMaxItem* function. Let's try to find out why it happens so. According to C++ rules the result of the subtraction of two pointers has type *ptrdiff_t*. When invoking *FindMaxItem* function the implicit conversion of *ptrdiff_t* type to *int* type occurs which will cause the loss of the high bits. This may be the reason for the incorrect program behavior while processing a large data size.

The correct solution will be to replace *int* type with *ptrdiff_t* type for it will allow to keep the whole range of values. The corrected code:

```
float FindMaxItem(float *array, ptrdiff_t arraySize) {
    float max = -FLT_MAX;
    for (ptrdiff_t i = 0; i != arraySize; ++i) {
        float item = *array++;
        if (max < item)
            max = item;
    }
    return max;
}
```

Analyzer tries as far as possible to recognize safe type conversions and keep from displaying warning messages on them. For example, the analyzer won't give a warning message on FindMaxItem function's call in the following code:

```
float Arr[1000];
float maxValue =
    FindMaxItem(Arr, sizeof(Arr)/sizeof(float));
```

When you are sure that the code is correct and the implicit type conversion of the actual function argument does not cause errors you may use the [explicit type conversion](#) so that to avoid showing warning messages. An example:

```
extern int nPenStyle
extern size_t nWidth;
extern COLORREF crColor;
...
// Call constructor CPen::CPen(int, int, COLORREF)
CPen myPen(nPenStyle, static_cast<int>(nWidth), crColor);
```

In that case if you suspect that the code contains incorrect explicit conversions of memsize types to 32-bit types about which the analyzer does not warn, you may use the [V202](#).

V108. Incorrect index type: 'foo[not a memsize-type]'. Use memsize type instead.

The analyzer found a possible error of indexing large arrays. The error may consist in the incorrect index determination.

The first example.

```
extern char *longString;
extern bool *isAlnum;
...
unsigned i = 0;
while (*longString) {
    isAlnum[i] = isalnum(*longString++);
    ++i;
}
```

The given code is absolutely correct for the 32-bit platform where it is actually impossible to process arrays more than *UINT_MAX* bytes (4Gb). On the 64-bit platform it is possible to process an array with the size more than 4 Gb that is sometimes very convenient. The error consists in the use of the variable of *unsigned* type for indexing the array *isAlnum*. When we fill the first *UINT_MAX* of the items the variable *i* overflow will occur and it will equal zero. As the result we'll begin to rewrite the array *isAlnum* items which are situated in the beginning and some items will be left unassigned.

The correction is to replace the variable *i* type with [memsize](#) type:

```
...
size_t i = 0;
while (*longString)
    isAlnum[i++] = isalnum(*longString++);
```

The second example.

```
class Region {
    float *array;
    int Width, Height, Depth;
    float Region::GetCell(int x, int y, int z) const;
    ...
};
```

```
float Region::GetCell(int x, int y, int z) const {
    return array[x + y * Width + z * Width * Height];
}
```

For computational modeling programs the main memory size is an important source, and the possibility to use more than 4 Gb of memory on the 64-bit architecture increases calculating possibilities greatly. In such programs one-dimensional arrays are often used which are then dealt with as three-dimensional ones. There are functions for that which similar to *GetCell* that provides access to the necessary items of the calculation area. But the given code may deal correctly with arrays containing not more than *INT_MAX* (2Gb) items. The reason is in the use of 32-bit *int* types which participate in calculating the item index. If the number of items in the array *array* exceeds *INT_MAX* (2 Gb) an overflow will occur and the index value will be determined incorrectly. Programmers often make a mistake trying to correct the code in the following way:

```
float Region::GetCell(int x, int y, int z) const {
    return array[static_cast<ptrdiff_t>(x) + y * Width +
                z * Width * Height];
}
```

They know that according to C++ rules the expression for calculating the index will have *ptrdiff_t* type and because of it hope to avoid the overflow. Unfortunately, the overflow may occur inside the subexpression $y * Width$ or $z * Width * Height$ for to determine them *int* type is still used.

If you want to correct the code without changing the types of the variables included into the expression you should convert each variable explicitly to memsize type:

```
float Region::GetCell(int x, int y, int z) const {
    return array[ptrdiff_t(x) +
                ptrdiff_t(y) * ptrdiff_t(Width) +
                ptrdiff_t(z) * ptrdiff_t(Width) *
                ptrdiff_t(Height)];
}
```

Another decision is to replace the variables types with memsize type:

```
class Region {
    float *array;
    ptrdiff_t Width, Height, Depth;
    float
    Region::GetCell(ptrdiff_t x, ptrdiff_t y, ptrdiff_t z) const;
    ...
};
float Region::GetCell(ptrdiff_t x, ptrdiff_t y, ptrdiff_t z) const
{
    return array[x + y * Width + z * Width * Height];
}
```

If you use expressions which type is different from memsize type for indexing but are sure about the code correctness, you may use the [explicit type conversion](#) to suppress the analyzer's warning messages as follows:

```
bool *Seconds;
int min, sec;
...
bool flag = Seconds[static_cast<size_t>(min * 60 + sec)];
```

If you suspect that the program may contain errors related to the incorrect explicit type conversion in expressions you may use the [V201](#).

The analyzer tries as far as possible to understand when using non-memsize-type as the array's index is safe and keep from displaying warnings in such cases. As the result the analyzer's behaviour can sometimes seem strange. In such situations we ask you not to hurry and try to analyze the situation. Let's consider the following code:

```
char Arr[] = { '0', '1', '2', '3', '4' };
char *p = Arr + 2;
cout << p[0u + 1] << endl;
cout << p[0u - 1] << endl; //V108
```

This code works correctly in 32-bit mode and displays numbers 3 and 1. While testing this code we'll get a warning message only on one line with the expression "p[0u - 1]". And it's absolutely right. If you compile and launch this example in 64-bit mode you'll see that the value 3 will be displayed and after that a program crash will occur.

The error relates to that indexing of "p[0u - 1]" is incorrect on a 64-bit system and this is what analyzer warns about. According to C++ rules "0u - 1" expression will have unsigned type and equal 0xFFFFFFFFu. On a 32-bit architecture addition of an index with this number will be the same as subtraction of 1. And on a 64-bit system 0xFFFFFFFFu value will be justly added to the index and memory will be addressed outside the array.

Of course indexing to arrays with the use of such types as int and unsigned is often safe. In this case analyzer's warnings may seem inappropriate. But you should keep in mind that such code still may be unsafe in case of its modernization for processing a different data set. The code with int and unsigned types can appear to be less efficient than it is possible on a 64-bit architecture.

If you are sure that indexation is correct you use "[Suppression of false alarms](#)" or use filters. You can use explicit type conversion in the code:

```
for (int i = 0; i != n; ++i)
    Array[static_cast<ptrdiff_t>(i)] = 0;
```

V109. Implicit type conversion of return value to memsize type.

The analyzer found a possible error related to the [implicit conversion](#) of the return value type. The error may consist in the incorrect determination of the return value.

Let's examine an example.

```
extern int Width, Height, Depth;
size_t GetIndex(int x, int y, int z) {
    return x + y * Width + z * Width * Height;
}
...
array[GetIndex(x, y, z)] = 0.0f;
```

If the code deals with large arrays (more than *INT_MAX* items) it will behave incorrectly and we will address not those items of the array *array* that we want. But the analyzer won't show a warning message on the line *array[GetIndex(x, y, z)] = 0.0f;* for it is absolutely correct. The analyzer informs about a possible error inside the function and is right for the error is located exactly there and is related to the arithmetic overflow. In spite of the fact that we return the type *size_t* value the expression $x + y * Width + z * Width * Height$ is determined with the use of type *int*.

To correct the error we should use the [explicit conversion](#) of all the variables included into the expression to [memsize](#) types.

```
extern int Width, Height, Depth;
size_t GetIndex(int x, int y, int z) {
    return (size_t)(x) +
        (size_t)(y) * (size_t)(Width) +
```



```

    (size_t) (z) * (size_t) (Width) * (size_t) (Height);
}

```

Another variant of correction is the use of other types for the variables included into the expression.

```

extern size_t Width, Height, Depth;
size_t GetIndex(size_t x, size_t y, size_t z) {
    return x + y * Width + z * Width * Height;
}

```

When you are sure that the code is correct and the implicit type conversion does not cause errors while porting to the 64-bit architecture you may use the explicit type conversion so that to avoid showing of the warning messages in this line. For example:

```

DWORD_PTR Calc(unsigned a) {
    return (DWORD_PTR) (10 * a);
}

```

In case you suspect that the code contains incorrect explicit type conversions to memsize types about which the analyzer does not show warnings you may use the [V201](#).

V110. Implicit type conversion of return value from memsize type to 32-bit type.

The analyzer found a possible error related to the implicit conversion of the return value. The error consists in dropping of the high bits in the 64-bit type which causes the loss of value.

Let's examine an example.

```

extern char *begin, *end;
unsigned GetSize() {
    return end - begin;
}

```

The result of the *end - begin* expression has type *ptrdiff_t*. But as the function returns type *unsigned* the implicit type conversion occurs which causes the loss of the result high bits. Thus, if the pointers *begin* and *end* refer to the beginning and the end of the array according to a larger *UINT_MAX* (4Gb), the function will return the incorrect value.

The correction consists in modifying the program in such a way so that the arrays sizes are kept and transported in [memsize](#) types. In this case the correct code of the *GetSize* function should look as follows:

```

extern char *begin, *end;
size_t GetSize() {
    return end - begin;
}

```

In some cases the analyzer won't display a warning message on type conversion if it is obviously correct. For example, the analyzer won't display a warning message on the following code where despite the fact that *sizeof()* operator's result is *size_t* type it can be safely placed into unsigned type:

```

unsigned GetSize() {
    return sizeof(double);
}

```

When you are sure that the code is correct and the implicit type conversion does not cause errors while porting to the 64-bit architecture you may use the explicit type conversion so that to avoid showing of the warning messages. For example:

```

unsigned GetBitCount() {
    return static_cast<unsigned>(sizeof(TypeRGBA) * 8);
}

```

If you suspect that the code contains incorrect explicit conversions of the return values types about which the analyzer does not warn you may use the [V202](#).

V111. Call of function 'foo' with variable number of arguments. N argument has memsize type.

The analyzer found a possible error related to the transfer of the actual argument of [memsize](#) type into the function with variable number of arguments. The possible error may consist in the change of demands made to the function on the 64-bit system.

Let's examine an example.

```

const char *invalidFormat = "%u";
size_t value = SIZE_MAX;
printf(invalidFormat, value);

```

The given code does not take into account that *size_t* type does not coincide with *unsigned* type on the 64-bit platform. It will cause the printing of the incorrect result in case if *value > UINT_MAX*. The analyzer warns you that memsize type is used as an actual argument. It means that you should check the line *invalidFormat* assigning the printing format. The correct variant may look as follows:

```

const char *validFormat = "%Iu";
size_t value = SIZE_MAX;
printf(validFormat, value);

```

In the code of a real application, this error can occur in the following form, e.g.:

```

wsprintf(szDebugMessage,
    _T("%s location %08x caused an access violation.\r\n"),
    readwrite,
    Exception->m_pAddr);

```

The second example.

```

char buf[9];
sprintf(buf, "%p", pointer);

```

The author of this inaccurate code did not take into account that the pointer size may exceed 32 bits later. As a result, this code will cause buffer overflow on the 64-bit architecture. After checking the code on which the V111 warning message is shown you may choose one of the two ways: to increase the buffer size or rewrite the code using safe constructions.

```

char buf[sizeof(pointer) + 1];
sprintf(buf, "%p", pointer);

```

```
// --- or ---
std::stringstream s;
s << pointer;
```

The third example.

```
char buf[9];
sprintf_s(buf, sizeof(buf), "%p", pointer);
```

While examining the second example you could rightly notice that in order to prevent the overflow you should use functions with security enhancements. In this case the buffer overflow won't occur but unfortunately the correct result won't be shown as well.

If the arguments types did not change their digit capacity the code is considered to be correct and warning messages won't be shown. The example:

```
printf("%d", 10*5);
CString str;
size_t n = sizeof(float);
str.Format(StrFormat, static_cast<int>(n));
```

Unfortunately, we often cannot distinguish the correct code from the incorrect one while diagnosing the described type of errors. This warning message will be shown on many of calls of the functions with variable items number even when the call is absolutely correct. It is related to the principal danger of using such C++ constructions. Most frequent problems are the problems with the use of variants of the following functions: *printf*, *scanf*, *CString::Format*. The generally accepted practice is to refuse them and to use safe programming methods. For example, you may replace *printf* with *cout* and *sprintf* with *boost::format* or *std::stringstream*.

V112. Dangerous magic number N used.

The analyzer found the use of a dangerous magic number. The possible error may consist in the use of numeric literal as special values or size of [memsize](#) type.

Let's examine the first example.

```
size_t ArraySize = N * 4;
size_t *Array = (size_t *)malloc(ArraySize);
```

A programmer while writing the program relied on that the size *size_t* will be always equal 4 and wrote the calculation of the array size "N * 4". This code dose not take into account that *size_t* on the 64-bit system will have 8 bytes and will allocate less memory than it is necessary. The correction of the code consists in the use of *sizeof* operator instead of a constant 4.

```
size_t ArraySize = N * sizeof(size_t);
size_t *Array = (size_t *)malloc(ArraySize);
```

The second example.

```
size_t n = static_cast<size_t>(-1);
if (n == 0xffffffffu) { ... }
```

Sometimes as an error code or other special marker the value "-1" is used which is written as "0xffffffff". On the 64-bit platform the written expression is incorrect and one should evidently use the value "-1".

```
size_t n = static_cast<size_t>(-1);
if (n == static_cast<size_t>(-1)) { ... }
```

Let's list magic numbers which may influence the efficiency of an application while porting it on the 64-bit system and due to this are diagnosed by analyzer.

Value:	Can be used as:
4	Number of bytes in type.
32	Number of bits in type.
0x7fffffff	Max value of signed variable. Mask for higher bit zero setting.
0x80000000	Min value of signed variable. Mask for higher bit selecting.
0xffffffff	Max value of unsigned variable. Alternative representation -1 as error indicator.

You should study the code thoroughly in order to see if there are magic constants and replace them with safe constants and expressions. For this purpose you may use *sizeof()* operator, special value from `<limits.h>`, `<inttypes.h>` etc.

In some cases magic constants are not considered unsafe. For example, there will be no warning on this code:

```
float Color[4];
```

V113. Implicit type conversion from memsize to double type or vice versa.

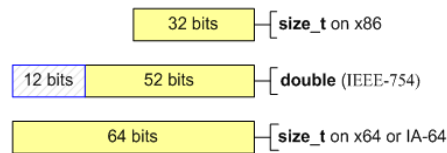
The analyzer found a possible error related to the [implicit conversion](#) of [memsize](#) type to *double* type of vice versa. The possible error may consist in the impossibility of storing the whole value range of memsize type in variables of *double* type.

Let's study an example.

```
SIZE_T size = SIZE_MAX;
double tmp = size;
size = tmp; // x86: size == SIZE_MAX
           // x64: size != SIZE_MAX
```

Double type has size 64 bits and is compatible IEEE-754 standard on 32-bit and 64-bit systems. Some programmers use *double* type to store and work with integer types.

The given example may be justified on a 32-bit system for *double* type has 52 significant bits and is capable to store a 32-bit integer value without a loss. But while trying to store an integer number in a variable of *double* type the exact value can be lost (see picture).



If an approximate value can be used for the work algorithm in your program no corrections are needed. But we would like to warn you about the results of the change of behavior of a code like this on 64-bit systems. In any case it is not recommended to mix integer arithmetic with floating point arithmetic.

V114. Dangerous explicit type pointer conversion.

The analyzer found a possible error related to the dangerous [explicit type conversion](#) of a pointer of one type to a pointer of another. The error may consist in the incorrect work with the objects to which the analyzer refers.

Let's examine an example. It contains the explicit type conversion of a *int* pointer to a *size_t* pointer.

```
int array[4] = { 1, 2, 3, 4 };
size_t *sizePtr = (size_t *) (array);
cout << sizePtr[1] << endl;
```

As you can see the result of the program output is different in 32-bit and 64-bit variants. On the 32-bit system the access to the array items is correct for the sizes of *size_t* and *int* types coincide and we see the output "2". On the 64-bit system we got "17179869187" in output for it is this value 17179869187 which stays in the first item of array *sizePtr*.

The correction of the situation described consists in refusing dangerous type conversions with the help of the program modernization. Another variant is to create a new array and to copy into it the values from the original array.

Of course not all the explicit conversions of pointer types are dangerous. In the following example the work result does not depend on the system capacity for *enum* type and *int* type have the same size on the 32-bit system and the 64-bit system as well. So the analyzer won't show any warning messages on this code.

```
int array[4] = { 1, 2, 3, 4 };
enum ENumbers { ZERO, ONE, TWO, THREE, FOUR };
ENumbers *enumPtr = (ENumbers *) (array);
cout << enumPtr[1] << endl;
```

V115. Memsized type is used for throw.

The analyzer found a possible error related to the use of [memsize](#) type for throwing an exception. The error may consist in the incorrect exception handling.

Let's examine an example of the code which contains *throw* and *catch* operators.

```
char *ptr1, *ptr2;
...
try {
    throw ptr2 - ptr1;
}
catch(int) {
    Foo();
}
```

On 64-bit system the exception handler will not work and the function *Foo ()* will not be called. This results from the fact that expression "*ptr2 - ptr1*" has type *ptrdiff_t* which on 64-bit system does not equivalent with type *int*.

The correction of the situation described consists in use of correct type for catch of exception. In this case is necessary use of *ptrdiff_t* type, as noted below.

```
try {
    throw ptr2 - ptr1;
}
catch(ptrdiff_t) {
    Foo();
}
```

More right correction will consist in refusal of similar practice of programming. We recommend to use special classes for sending information about the error.

V116. Memsized type is used for catch.

The analyzer found a possible error related to the use of [memsize](#) type for catching exception. The error may consist in the incorrect exception handling.

Let's examine an example of the code which contains *throw* and *catch* operators.

```
try {
    try {
        throw UINT64(-1);
    }
    catch(size_t) {
        cout << "x64 portability issues" << endl;
    }
}
catch(UINT64) {
    cout << "OK" << endl;
}
```

The work result on the 32-bit system: OK

The work result on the 64-bit system: x64 portability issues

This behavior change is connected with what on 64-bit system the *size_t* type is equivalent to *UINT64*.

Correction of the described situation consists in change of a code for achievement of necessary logic of work.

More right correction will consist in refusal of similar practice of programming. We recommend using special classes for sending information about the error.

V117. Memsized type is used in the union.

The analyzer found a possible error related to the use of [memsize](#) inside a union. The error may occur while working with such unions without taking into account the size changes of memsize types on the 64-bit system.

One should be attentive to the unions which contain pointers and other members of memsize type.

The first example.

Sometimes one needs to work with a pointer as with an integer. The code in the example is convenient because the [explicit type conversions](#) are not used for work with the pointer number form.

```
union PtrNumUnion {
    char *m_p;
    unsigned m_n;
} u;
...
u.m_p = str;
u.m_n += delta;
```

This code is correct on 32-bit systems and is incorrect on 64-bit ones. Changing the *m_n* member on the 64-bit system we work only with a part of the *m_p* pointer. One should use that type which would conform with the pointer size as follows.

```
union PtrNumUnion {
    char *m_p;
    size_t m_n; //type fixed
} u;
```

The second example.

Another frequent case of use of a union is the representation of one member as a set of smaller ones. For example, we may need to split the *size_t* type value into bytes for realization of the table algorithm of counting zero bits in a byte.

```
union SisetToBytesUnion {
    size_t value;
    struct {
        unsigned char b0, b1, b2, b3;
    } bytes;
} u;

SisetToBytesUnion u;
u.value = value;
size_t zeroBitsN = TranslateTable[u.bytes.b0] +
    TranslateTable[u.bytes.b1] +
    TranslateTable[u.bytes.b2] +
    TranslateTable[u.bytes.b3];
```

A fundamental algorithmic error is made here which is based on the supposition that the *size_t* type consists of 4 bytes. The automatic search of algorithmic errors is not possible on the current stage of development of static analyzers but Viva64 provides search of all the unions which contain memsize types. Looking through the list of such potentially dangerous unions a user can find logical errors. On finding the union given in the example a user can detect an algorithmic error and rewrite the code in the following way.

```
union SisetToBytesUnion {
    size_t value;
    unsigned char bytes[sizeof(value)];
} u;

SisetToBytesUnion u;
u.value = value;
size_t zeroBitsN = 0;
for (size_t i = 0; i != sizeof(bytes); ++i)
    zeroBitsN += TranslateTable[bytes[i]];
```

This warning message is similar to the warning [V122](#).

V118. malloc() function accepts a dangerous expression in the capacity of an argument.

The analyzer detected a potential error relating to using a dangerous expression serving as an actual argument for malloc function. The error may lie in incorrect suggestions about types' sizes defined as numerical constants.

The analyzer considers suspicious those expressions which contain constant literals multiple of four but which lack sizeof() operator.

Example 1.

An incorrect code of memory allocation for a matrix 3x3 of items of *size_t* type may look as follows:

```
size_t *pMatrix = (size_t *)malloc(36); // V118
```

Although this code could work very well in a 32-bit system, using number 36 is incorrect. When compiling a 64-bit version 72 bytes must be allocated. You may use sizeof() operator to correct this error:

```
size_t *pMatrix = (size_t *)malloc(9 * sizeof(size_t));
```

Example 2.

The following code based on the suggestion that the size of Item structure is 12 bytes is also incorrect for a 64-bit system:

```
struct Item {
    int m_a;
    int m_b;
    Item *m_pParent;
};
Item *items = (Item *)malloc(GetArraySize() * 12); // V118
```

Correction of this error also consists in using sizeof() operator to correctly calculate the size of the structure:

```
Item *items = (Item *)malloc(GetArraySize() * sizeof(Item));
```

These errors are simple and easy to correct. But they are nevertheless dangerous and difficult to find in case of large applications. That's why diagnosis of such errors is implemented as a separate rule.

Presence of a constant in an expression which is a parameter for malloc() function does not necessarily means that V118 warning will be always shown on it. If sizeof() operator participates in the expression this construction is safe. Here is an example of a code which the analyzer considers safe:

```
int *items = (int *)malloc(sizeof(int) * 12);
```

V119. More than one sizeof() operator is used in one expression.

The analyzer detected an unsafe arithmetic expression containing several sizeof() operators. Such expressions can potentially contain errors relating to incorrect calculations of the structures' sizes without taking into account field alignment.

Example:

```
struct MyBigStruct {
    unsigned m_numberOfPointers;
    void *m_Pointers[1];
};
size_t n2 = 1000;
void *p;
p = malloc(sizeof(unsigned) + n2 * sizeof(void *));
```

To calculate the size of the structure which will contain 1000 pointers, an arithmetic expression is used which is correct at first sight. The sizes of the base types are defined by sizeof() operators. It is good but not sufficient for correct calculation of the necessary memory size. You should also take into account field alignment.

This example is correct for a 32-bit mode for the sizes of the pointers and unsigned type coincide. They are both 4 bytes. The pointers and unsigned type are aligned also at the boundary of four bytes. So the necessary memory size will be calculated correctly.

In a 64-bit code the size of the pointer is 8 bytes. Pointers are aligned at the boundary of 8 bytes as well. It leads to that after m_numberOfPointers variable 4 additional bytes will be situated at the boundary of 8 bytes to align the pointers.

To calculate the correct size you should use offsetof function:

```
p = malloc(offsetof(MyBigStruct, m_Pointers) +
            n * sizeof(void *));
```

In many cases using several sizeof() operators in one expression is correct and the analyzer ignores such constructions. Here is an example of safe expressions with several sizeof operators:

```
int MyArray[] = { 1, 2, 3 };
size_t MyArraySize =
    sizeof(MyArray) / sizeof(MyArray[0]);
assert(sizeof(unsigned) < sizeof(size_t));
size_t strLen = sizeof(String) - sizeof(TCHAR);
```

V120. Member operator[] of object 'foo' declared with 32-bit type argument, but called with memsize type argument.

The analyzer detected a potential error of working with classes that contain operator[]. Classes with an overloaded operator[] are usually a kind of an array where the index of the item being called is operator[] argument. If operator[] has a 32-bit type formal argument but [memsize](#)-type is used as an actual argument, it might indicate an error. Let us consider an example leading to the warning V120:

```
class MyArray {
    int m_arr[10];
public:
    int &operator[](unsigned i) { return m_arr[i]; }
} Object;
size_t k = 1;
Object[k] = 44; //V120
```

This example does not contain an error but might indicate an architecture shortcoming. You should either work with MyArray using 32-bit indexes or modify operator[] so that it takes an argument of size_t type. The latter is preferable because memsize-types not only serve to make a program safer but sometimes allow the compiler to build a more efficient code.

The related diagnostic warnings are [V108](#) and [V302](#).

V121. Implicit conversion of the type of 'new' operator's argument to size_t type.

The analyzer detected a potential error related to calling the operator new. A value of a non-[memsize](#) type is passed to the operator "new" as an argument. The operator new takes values of the type size_t, and passing a 32-bit actual argument may signal a potential overflow that may occur when calculating the memory amount being allocated. Here is an example:

```
unsigned a = 5;
unsigned b = 1024;
unsigned c = 1024;
unsigned d = 1024;
char *ptr = new char[a*b*c*d]; //V121
```

Here you may see an overflow occurring when calculating the expression "a*b*c*d". As a result, the program allocates less memory than it should. To correct the code, use the type size_t:

```
size_t a = 5;
size_t b = 1024;
size_t c = 1024;
size_t d = 1024;
char *ptr = new char[a*b*c*d]; //Ok
```

The following code also contains an error:

```
#define x 1024
#define y 1024
#define z 1024
#ifdef _M_X64
#define n 5
#else
#define n 1
#endif
char *p = new char[x*y*z*n]; //V121
```

This error is diagnosed only when the option MoreThan2Gb in the analyzer's settings is set to "true". The error will not be diagnosed if the value of the argument is defined as a safe 32-bit constant value. Here is an example of safe code:

```
char *ptr = new char[100];
```

```
const int size = 3*3;
char *p2 = new char[size];
```

This warning message is similar to the warning [V106](#).

V122. Memsized type is used in the struct/class.

Sometimes you might need to find all the fields in the structures that have a [memsize](#)-type. You can find such fields using the V122 diagnostic rule.

The necessity to view all the memsize-fields might appear when you port a program that has structure serialization, for example, into a file. Consider an example:

```
struct Header
{
    unsigned m_version;
    size_t m_bodyLen;
};
...
size_t size = fwrite(&header, 1, sizeof(header), file);
...
```

This code writes a different number of bytes into the file depending on the mode it is compiled in - either Win32 or Win64. This might violate compatibility of files' formats or cause other errors.

The task of automating the detection of such errors is almost impossible to solve. However, if there are some reasons to suppose that the code might contain such errors, developers can once check all the structures that participate in serialization. It is for this purpose that you may need a check with the V122 rule. By default it is disabled since it generates false warnings in more than 99% of cases.

In the example above, the V122 message will be produced on the line "size_t m_bodyLen;". To correct this code, you may use types of fixed size:

```
struct Header
{
    My_UInt32 m_version;
    My_UInt32 m_bodyLen;
};
...
size_t size = fwrite(&header, 1, sizeof(header), file);
...
```

Let's consider other examples where the V122 message will be generated:

```
class X
{
    int i;
    DWORD_PTR a; //V122
    DWORD_PTR b[3]; //V122
    float c[3][4];
    float *ptr; //V122
};
```

[V117](#) is a related diagnostic message.

V123. Allocation of memory by the pattern "(X*)malloc(sizeof(Y))" where the sizes of X and Y types are not equal.

The analyzer found a potential error related to the operation of memory allocation. When calculating the amount of memory to be allocated, the sizeof(X) operator is used. The result returned by the memory allocation function is converted to a different type, "(Y *)", instead of "(X *)". It may indicate allocation of insufficient or excessive amount of memory.

Consider the first example:

```
int **ArrayOfPointers = (int **)malloc(n * sizeof(int));
```

The misprint in the 64-bit program here will cause allocation of memory twice less than necessary. In the 32-bit program, the sizes of the "int" type and "pointer to int" coincide and the program works correctly despite the misprint.

This is the correct version of the code:

```
int **ArrayOfPointers = (int **)malloc(n * sizeof(int *));
```

Consider another example where more memory is allocated than needed:

```
unsigned *p = (unsigned *)malloc(len * sizeof(size_t));
```

A program with such code will most probably work correctly both in the 32-bit and 64-bit versions. But in the 64-bit version, it will allocate more memory than it needs. This is the correct code:

```
unsigned *p = (unsigned *)malloc(len * sizeof(unsigned));
```

In some cases the analyzer does not generate a warning although the types X and Y do not coincide. Here is an example of such correct code:

```
BYTE *simpleBuf = (BYTE *)malloc(n * sizeof(float));
```

V124. Function 'Foo' writes/reads 'N' bytes. The alignment rules and type sizes have been changed. Consider reviewing this value.

The analyzer detected a potential error: the size of data being written or read is defined by a constant. When the code is compiled in the 64-bit mode, the sizes of some data and their alignment boundaries will change. The sizes of base types and their alignment boundaries are shown in the picture:

Type	32-bit x86 Windows		64-bit x86-64 Windows		32-bit x86 GNU/Linux		64-bit x86-64 GNU/Linux	
	sizeof	alignof	sizeof	alignof	sizeof	alignof	sizeof	alignof
bool	1	1	1	1	1	1	1	1
wchar_t	2	2	2	2	4	4	4	4
short	2	2	2	2	2	2	2	2
int	4	4	4	4	4	4	4	4
long	4	4	4	4	4	4	8	8
size_t	4	4	8	8	4	4	8	8
ptrdiff_t	4	4	8	8	4	4	8	8
intptr_t	4	4	8	8	4	4	8	8
uintptr_t	4	4	8	8	4	4	8	8
double	8	8	8	8	8	4	8	8
long double	8	8	8	8	12	4	16	16
void*	4	4	8	8	4	4	8	8

The analyzer examines code fragments where the size of data being written or read is defined explicitly. The programmer must review these fragments. Here is a code sample:

```
size_t n = fread(buf, 1, 40, f_in);
```

Constant 40 may be an incorrect value from the viewpoint of the 64-bit system. Perhaps you should write it so:

```
size_t n = fread(buf, 1, 10 * sizeof(size_t), f_in);
```

V125. It is not advised to declare type 'T' as 32-bit type.

The analyzer detected a potential error: 64-bit code contains definitions of reserved types, the latter being defined as 32-bit ones. For example:

```
typedef unsigned size_t;
typedef __int32 INT_PTR;
```

Such type definitions may cause various errors since these types have different sizes in different parts of the program and libraries. For instance, the `size_t` type is defined in the `stddef.h` header file for the C language and in the `cstdint.h` file for the C++ language.

References:

1. Knowledge Base. Is there a way to make the type `size_t` 32-bit in a 64-bit program? <http://www.viva64.com/en/k/0021/>
2. Knowledge Base. Is `size_t` a standard type in C++? And in C? <http://www.viva64.com/en/k/0022/>

V126. Be advised that the size of the type 'long' varies between LLP64/LP64 data models.

This diagnostic message lets you find all the 'long' types used in a program.

Of course, presence of the 'long' type in a program is not an error in itself. But you may need to review all the fragments of the program text where this type is used when you create portable 64-bit code that must work well in Windows and Linux.

Windows and Linux use different data models for the 64-bit architecture. A data model means correlations of sizes of base data types such as `int`, `float`, `pointer`, etc. Windows uses the LLP64 data model while Linux uses the LP64 data model. In these models, the sizes of the 'long' type are different.

In Windows (LLP64), the size of the 'long' type is 4 bytes.

In Linux (LP64), the size of the 'long' type is 8 bytes.

The difference of the 'long' type's sizes may make files' formats incompatible or cause errors when developing code executed in Linux and Windows. So if you want, you may use PVS-Studio to review all the code fragments where the 'long' type is used.

References:

1. Terminology. Data model. <http://www.viva64.com/en/t/0012/>

V127. An overflow of the 32-bit variable is possible inside a long cycle which utilizes a memsize-type loop counter.

The analyzer detected a potential error: a 32-bit variable might overflow in a long loop. Of course, the analyzer will not be able to find all the possible cases when variable overflows in loops occur. But it will help you find some incorrect type constructs. For example:

```
int count = 0;
```

```

for (size_t i = 0; i != N; i++)
{
    if ((A[i] & MASK) != 0)
        count++;
}

```

This code works well in a 32-bit program. The variable of the 'int' type is enough to count the number of some items in the array. But in a 64-bit program the number of these items may exceed INT_MAX and an overflow of the 'count' variable will occur. This is what the analyzer warns you about by generating the V127 message. This is the correct code:

```

size_t count = 0;
for (size_t i = 0; i != N; i++)
{
    if ((A[i] & MASK) != 0)
        count++;
}

```

The analyzer also contains several additional checks to make false reports fewer. For instance, the V127 warning will not be generated when we deal with a short loop. Here you are a sample of code the analyzer considers safe:

```

int count = 0;
for (size_t i = 0; i < 100; i++)
{
    if ((A[i] & MASK) != 0)
        count++;
}

```

V201. Explicit conversion from 32-bit integer type to memsize type.

It informs about the presence of the [explicit type conversion](#) from 32-bit integer type to [memsize](#) type which may hide one of the following errors: [V101](#), [V102](#), [V104](#), [V105](#), [V106](#), [V108](#), [V109](#). You may address to the given warnings list to find out the cause of showing the diagnosis message V201.

The V201 warning applied to conversions of 32-bit integer types to pointers before. Such conversions are rather dangerous, so we singled them out into a separate diagnostic rule [V204](#).

Keep in mind that most of the warnings of this type will be likely shown on the correct code. Here are some examples of the correct and incorrect code on which this warning will be shown.

The examples of the incorrect code.

```

int i;
ptrdiff_t n;
...
for (i = 0; (ptrdiff_t) i) != n; ++i) {    //V201
    ...
}

unsigned width, height, depth;
...
size_t arraySize = size_t(width * height * depth);    //V201

```

The examples of the correct code.

```

const size_t seconds = static_cast<size_t>(60 * 60);    //V201
unsigned *array;
...
size_t sum = 0;
for (size_t i = 0; i != n; i++) {
    sum += static_cast<size_t>(array[i] / 4);    //V201
}
unsigned width, height, depth;
...
size_t arraySize =
size_t(width) * size_t(height) * size_t(depth);    //V201

```

V202. Explicit conversion from memsize type to 32-bit integer type.

It informs about the presence of the explicit integer [memsize](#) type conversion to 32-bit type which may hide one of the following errors: [V103](#), [V107](#), [V110](#). You may see the given warnings list to find out the cause of showing the warning message V202.

The V202 warning applied to conversions of pointers to 32-bit integer types before. Such conversions are rather dangerous, so we singled them out into a separate rule [V205](#).

Keep in mind that most of the warnings of this type will be likely shown on the correct code. Here are some examples of the correct and incorrect code on which this warning will be shown.

The examples of the incorrect code.

```

size_t n;
...
for (unsigned i = 0; i != (unsigned)n; ++i) {    //V202
    ...
}

UINT_PTR width, height, depth;
...
UINT arraySize = UINT(width * height * depth);    //V202

```

The examples of the correct code.

```

const unsigned bits =
    unsigned(sizeof(object) * 8);    //V202

extern size_t nPane;
extern HICON hIcon;
BOOL result =
    SetIcon(static_cast<int>(nPane), hIcon);    //V202

```

V203. Explicit type conversion from memsize to double type or vice versa.

The analyzer found a possible error related to the explicit conversion of [memsize](#) type into *double* type and vice versa. The possible error may consist in the impossibility to save the whole range of values of [memsize](#) type in variables of *double* type.

This error is completely similar to error [V113](#). The difference is in that the [explicit type conversion](#) is used as in a further example:

```
SIZE_T size = SIZE_MAX;
double tmp = static_cast<double>(size);
size = static_cast<SIZE_T>(tmp); // x86: size == SIZE_T
                                // x64: size != SIZE_T
```

To study this kind of errors see the description of error [V113](#).

V204. Explicit conversion from 32-bit integer type to pointer type.

This warning informs you about an [explicit conversion](#) of a 32-bit integer type to a pointer type. We used the [V201](#) diagnostic rule before to diagnose this situation. But explicit conversion of the 'int' type to pointer is much more dangerous than conversion of 'int' to 'intptr_t'. That is why we created a separate rule to search for explicit type conversions when handling pointers.

Here is a sample of incorrect code.

```
int n;
float *ptr;
...
ptr = (float *) (n);
```

The 'int' type's size is 4 bytes in a 64-bit program, so it cannot store a pointer whose size is 8 bytes. Type conversion like in the sample above usually signals an error.

What is very unpleasant about such errors is that they can hide for a long time before you reveal them. A program might store pointers in 32-bit variables and work correctly for some time as long as all the objects created in the program are located in low-order addresses of memory.

If you need to store a pointer in an integer variable for some reason, you'd better use [memsize-types](#). For instance: `size_t`, `ptrdiff_t`, `intptr_t`, `uintptr_t`.

This is the correct code:

```
intptr_t n;
float *ptr;
...
ptr = (float *) (n);
```

However, there is a specific case when you may store a pointer in 32-bit types. I am speaking about handles which are used in Windows to work with various system objects. Here are examples of such types: `HANDLE`, `HWND`, `HMENU`, `HPALETTE`, `HBITMAP`, etc. Actually these types are pointers. For instance, `HANDLE` is defined in header files as "typedef void *HANDLE;".

Although handles are 64-bit pointers, only the less significant 32 bits are employed in them for the purpose of better compatibility (for example, to enable 32-bit and 64-bit processes interact with each other). For details, see "[Microsoft Interface Definition Language \(IDL\): 64-Bit Porting Guide](#)" (USER and GDI handles are sign extended 32b values).

Such pointers can be stored in 32-bit data types (for instance, `int`, `DWORD`). To cast such pointers to 32-bit types and vice versa [special functions](#) are used:

```
void * Handle64ToHandle( const void * POINTER_64 h )
void * POINTER_64 HandleToHandle64( const void *h )
long HandleToLong ( const void *h )
unsigned long HandleToUlong ( const void *h )
void * IntToPtr ( const int i )
void * LongToHandle ( const long h )
void * LongToPtr ( const long l )
void * Ptr64ToPtr ( const void * POINTER_64 p )
int PtrToInt ( const void *p )
long PtrToLong ( const void *p )
void * POINTER_64 PtrToPtr64 ( const void *p )
short PtrToShort ( const void *p )
unsigned int PtrToUint ( const void *p )
unsigned long PtrToUlong ( const void *p )
unsigned short PtrToUshort ( const void *p )
void * UIntToPtr ( const unsigned int ui )
void * ULongToPtr ( const unsigned long ul )
```

Additional materials on this topic:

1. Knowledge Base. How to correctly cast a pointer to int in a 64-bit application? <http://www.viva64.com/en/k/0005/>
2. 32-bit pointers in a 64-bit world. <http://www.viva64.com/en/r/0106/>

V205. Explicit conversion of pointer type to 32-bit integer type.

This warning informs you about an [explicit conversion](#) of a pointer type to a 32-bit integer type. We used the [V202](#) diagnostic rule before to diagnose this situation. But explicit conversion of a pointer to the 'int' type is much more dangerous than conversion of 'intptr_t' to 'int'. That is why we created a separate rule to search for explicit type conversions when handling pointers.

Here is a sample of incorrect code.

```
int n;
float *ptr;
...
n = (int)ptr;
```

The 'int' type's size is 4 bytes in a 64-bit program, so it cannot store a pointer whose size is 8 bytes. Type conversion like in the sample above usually signals an error.

What is very unpleasant about such errors is that they can hide for a long time before you reveal them. A program might store pointers in 32-bit variables and work correctly for some time as long as all the objects created in the program are located in low-order addresses of memory.

If you need to store a pointer in an integer variable for some reason, you'd better use [memsize-types](#). For instance: `size_t`, `ptrdiff_t`, `intptr_t`, `uintptr_t`.

This is the correct code:

```
intptr_t n;
float *ptr;
...
n = (intptr_t)ptr;
```

However, there is a specific case when you may store a pointer in 32-bit types. I am speaking about handles which are used in Windows to work with various system objects. Here are examples of such types: HANDLE, HWND, HMENU, HPALETTE, HBITMAP, etc. Actually these types are pointers. For instance, HANDLE is defined in header files as "typedef void *HANDLE;".

Although handles are 64-bit pointers, only the less significant 32 bits are employed in them for the purpose of better compatibility (for example, to enable 32-bit and 64-bit processes interact with each other). For details, see "[Microsoft Interface Definition Language \(IDL\): 64-Bit Porting Guide](#)" (USER and GDI handles are sign extended 32b values).

Such pointers can be stored in 32-bit data types (for instance, int, DWORD). To cast such pointers to 32-bit types and vice versa [special functions](#) are used:

```
void          * Handle64ToHandle( const void * POINTER_64 h )
void * POINTER_64 HandleToHandle64( const void *h )
long          HandleToLong      ( const void *h )
unsigned long HandleToUlong     ( const void *h )
void          * IntToPtr        ( const int i )
void          * LongToHandle    ( const long h )
void          * LongToPtr       ( const long l )
void          * Ptr64ToPtr      ( const void * POINTER_64 p )
int           PtrToInt          ( const void *p )
long          PtrToLong         ( const void *p )
void * POINTER_64 PtrToPtr64    ( const void *p )
short         PtrToShort        ( const void *p )
unsigned int   PtrToUint         ( const void *p )
unsigned long  PtrToUlong        ( const void *p )
unsigned short PtrToUshort       ( const void *p )
void          * UIntToPtr       ( const unsigned int ui )
void          * ULONGToPtr      ( const unsigned long ul )
```

Additional materials on this topic:

1. Knowledge Base. How to correctly cast a pointer to int in a 64-bit application? <http://www.viva64.com/en/k/0005/>
2. 32-bit pointers in a 64-bit world. <http://www.viva64.com/en/r/0106/>

V220. Suspicious sequence of types castings: memsize -> 32-bit integer -> memsize.

The warning informs you about a strange sequence of type conversions. A [memsize](#)-type is explicitly cast to a 32-bit integer type and then is again cast to a memsize-type either explicitly or implicitly. Such a sequence of conversions leads to a loss of high-order bits. Usually it signals a crucial error.

Consider this sample:

```
char *p1;
char *p2;
ptrdiff_t n;
...
n = int(p1 - p2);
```

We have an unnecessary conversion to the 'int' type here. It must not be here and even might cause a failure if p1 and p2 pointers are more than INT_MAX items away from each other in a 64-bit program.

This is the correct code:

```
char *p1;
char *p2;
ptrdiff_t n;
...
n = p1 - p2;
```

Let's consider another sample:

```
BOOL SetItemData(int nItem, DWORD_PTR dwData);
...
CItemData *pData = new CItemData;
...
CListCtrl::SetItemData(nItem, (DWORD)pData);
```

This code will cause an error if the CItemData object is created beyond the four low-order Gbytes of memory. This is the correct code:

```
BOOL SetItemData(int nItem, DWORD_PTR dwData);
...
CItemData *pData = new CItemData;
...
CListCtrl::SetItemData(nItem, (DWORD_PTR)pData);
```

V301. Unexpected function overloading behavior. See N argument of function 'foo' in derived class 'derived' and base class 'base'.

The analyzer found a possible error related to the changes in the overriding virtual functions behavior.

The example of the change in the virtual function behavior.

```
class CWinApp {
...
    virtual void WinHelp(DWORD_PTR dwData, UINT nCmd);
...
};
class CSampleApp : public CWinApp {
...
    virtual void WinHelp(DWORD dwData, UINT nCmd);
...
};
```

It is the common example which the developer may face while porting his application to the 64-bit architecture. Let's follow the life-cycle of the developing of some application. Suppose it was being developed for Visual Studio 6.0. at first when the function *WinHelp* in class *CWinApp* had the following prototype:

```
virtual void WinHelp(DWORD dwData, UINT nCmd = HELP_CONTEXT);
```

It would be absolutely correct to implement the overlap of the virtual function in class *CSampleApp*, as it is shown in the example. Then the project was placed into Visual Studio 2005 where the prototype of the function in class *CWinApp* underwent changes that consist in replacing *DWORD* type with *DWORD_PTR* type. On the 32-bit platform this program will continue to work properly for here *DWORD* and *DWORD_PTR* types coincide. Troubles will occur while compiling this code

for the 64-bit platform. We get two functions with the same names but with different parameters the result of which is that the user's code won't be called.

The analyzer allows to find such errors the correction of which is not difficult. It is enough to change the function prototype in the successor class as follows:

```
class CSampleApp : public CWinApp {
...
    virtual void WinHelp(DWORD_PTR dwData, UINT nCmd);
...
};
```

V302. Member operator[] of 'foo' class has a 32-bit type argument. Use memsize-type here.

The analyzer detected a potential error of working with classes that contain operator[]. Classes with an overloaded operator[] are usually a kind of an array where the index of the item being called is operator[] argument. If operator[] has a 32-bit type argument it might indicate an error. Let us consider an example leading to the warning V302:

```
class MyArray {
    std::vector<int> m_arr;
...
    float &operator[](int i) //V302
    {
        DoSomething();
        return m_arr[i];
    }
} A;
...
int x = 2000;
int y = 2000;
int z = 2000;
A[x * y * z] = 33;
```

If the class is designed to work with many arguments, implementing operator[] like this is incorrect because it does not allow addressing the items whose numbers are more than UINT_MAX. To diagnose the error in the example above you should point to the potentially incorrect operator[]. The expression "x * y * z" does not look suspicious because there is no implicit type conversion. When we correct operator[] in the following way:

```
float &operator[](ptrdiff_t i);
```

PVS-Studio analyzer warns about a potential error in the line "A[x * y * z] = 33;" and now we can make the code absolutely correct. Here is an example of the corrected code:

```
class MyArray {
    std::vector<int> m_arr;
...
    float &operator[](ptrdiff_t i) //V302
    {
        DoSomething();
        return m_arr[i];
    }
} A;
...
ptrdiff_t x = 2000;
ptrdiff_t y = 2000;
ptrdiff_t z = 2000;
A[x * y * z] = 33;
```

The related diagnostic warnings are [V108](#) and [V120](#).

V303. The function is deprecated in the Win64 system. It is safer to use the 'foo' function.

[EnumProcessModules](#)

[SetWindowLong](#)

[GetFileSize](#)

You should replace some functions with their new versions when porting an application to 64-bit systems. Otherwise, the 64-bit application might work incorrectly. The analyzer warns about the use of deprecated functions in code and offers versions to replace them.

Let's consider several examples of deprecated functions:

EnumProcessModules

Extract from MSDN:

To control whether a 64-bit application enumerates 32-bit modules, 64-bit modules, or both types of modules, use the **EnumProcessModulesEx** function.

SetWindowLong

Extract from MSDN:

This function has been superseded by the **SetWindowLongPtr** function. To write code that is compatible with both 32-bit and 64-bit versions of Windows, use the **SetWindowLongPtr** function.

GetFileSize

Extract from MSDN:

When **lpFileSizeHigh** is **NULL**, the results returned for large files are ambiguous, and you will not be able to determine the actual size of the file. It is recommended that you use **GetFileSizeEx** instead.

V501. There are identical sub-expressions to the left and to the right of

the 'foo' operator.

The analyzer found a code fragment that most probably has a logic error. There is an operator (<, >, <=, >=, ==, !=, &&, ||, -, /) in the program text to the left and to the right of which there are identical subexpressions.

Consider an example:

```
if (a.x != 0 && a.x != 0)
```

In this case, the '&&' operator is surrounded by identical subexpressions "a.x != 0" and it allows us to detect an error made through inattention. The correct code that will not look suspicious to the analyzer looks in the following way:

```
if (a.x != 0 && a.y != 0)
```

Consider another example of an error detected by the analyzer in the code of a real application:

```
class Foo {
    int iChilds[2];
    ...
    bool hasChilds() const { return(iChilds > 0 || iChilds > 0); }
    ...
}
```

In this case, the code is senseless though it is compiled successfully and without any warnings. Correct code must look as follows:

```
bool hasChilds() const { return(iChilds[0] > 0 || iChilds[1] > 0);}
```

The analyzer does not generate the warning in all the cases when there are identical subexpressions to the left and to the right of the operator.

The first exception refers to those constructs where the increment operator ++, the decrement operator - or += and -= operator are used. Here is an example taken from a real application:

```
do {
} while (++scan == ++match && ++scan == ++match &&
        ++scan == ++match && ++scan == ++match &&
        ++scan == ++match && ++scan == ++match &&
        ++scan == ++match && ++scan == ++match &&
        scan < strend());
```

The analyzer considers this code safe.

The second exception refers to comparison of two equal numbers. Programmers often employ this method to disable some program branches. Here is an example:

```
#if defined(_OPENMP)
#include <omp.h>
#else
#define omp_get_thread_num() 0
...
#endif
...
if (0 == omp_get_thread_num()) {
```

The last exception refers to comparison that uses macros:

```
#define _WINVER_NT4_ 0x0004
#define _WINVER_95_ 0x0004
...
UINT winver = g_App.m_pPrefs->GetWindowsVersion();
if(winver == _WINVER_95_ || winver == _WINVER_NT4_)
```

You should keep in mind that the analyzer might generate a warning on a correct construct in some cases. For instance, the analyzer does not consider side effects when calling functions:

```
if (wr.readChar() == '\0' && wr.readChar() == '\0')
```

Another example of a false alarm was noticed during unit-tests of some project - in the part of it where the correctness of the overloaded operator '==' was checked:

```
CHECK(VDStringA() == VDStringA(), true);
CHECK(VDStringA("abc") == VDStringA("abc"), true);
```

The diagnostic message isn't generated if two identical expressions of 'float' or 'double' types are being compared. Such a comparison allows to identify the value as NaN. The example of code implementing the verification of this kind:

```
bool isnan(double X) { return X != X; }
```

V502. Perhaps the '?:' operator works in a different way than it was expected. The '?:' operator has a lower priority than the 'foo' operator.

The analyzer found a code fragment that most probably has a logic error. The program text has an expression that contains the ternary operator '?:' and might be calculated in a different way than the programmer expects.

The '?:' operator has a lower priority than operators ||, &&, |, ^, &, !=, ==, >=, <=, >, <, >>, <<, -, +, %, /, *. One might forget about it and write an incorrect code like the following one:

```
bool bAdd = ...;
size_t rightLen = ...;
size_t newTypeLen = rightLen + bAdd ? 1 : 0;
```

Having forgotten that the '+' operator has a higher priority than the '?:' operator, the programmer expects that the code is equivalent to "rightLen + (bAdd ? 1 : 0)". But actually the code is equivalent to the expression "(rightLen + bAdd) ? 1 : 0".

The analyzer diagnoses the probable error by checking:

- 1) If there is a variable or subexpression of the bool type to the left of the '?:' operator.
- 2) If this subexpression is compared to / added to / multiplied by... the variable whose type is other than bool.

If these conditions hold, it is highly probable that there is an error in this code and the analyzer will generate the warning message we are discussing.

Here are some other examples of incorrect code:

```
bool b;
int x, y, z, h;
...
x = y < b ? z : h;
x = y + (z != h) ? 1 : 2;
```

The programmer most likely wanted to have the following correct code:

```
bool b;
int x, y, z, h;
...
x = y < (b ? z : h);
x = y + ((z != h) ? 1 : 2);
```

If there is a type other than `bool` to the left of the `'?:'` operator, the analyzer thinks that the code is written in the C style (where there is no `bool`) or that it is written using class objects and therefore the analyzer cannot find out if this code is dangerous or not.

Here is an example of correct code in the C style that the analyzer considers correct too:

```
int conditions1;
int conditions2;
int conditions3;
...
char x = conditions1 + conditions2 + conditions3 ? 'a' : 'b';
```

V503. This is a nonsensical comparison: pointer < 0.

The analyzer found a code fragment that has a nonsensical comparison. It is most probable that this code has a logic error. Here is an example:

```
class MyClass {
public:
    CObj *Find(const char *name);
    ...
} Storage;

if (Storage.Find("foo") < 0)
    ObjectNotFound();
```

It seems almost incredible that such a code can exist in a program. However, the reason for its appearance might be quite simple. Suppose we have the following code in our program:

```
class MyClass {
public:
    // If the object is not found, the function
    // Find() returns -1.
    ptrdiff_t Find(const char *name);
    CObj *Get(ptrdiff_t index);
    ...
} Storage;
...
ptrdiff_t index = Storage.Find("ZZ");
if (index >= 0)
    Foo(Storage.Get(index));
...
if (Storage.Find("foo") < 0)
    ObjectNotFound();
```

This is correct yet not very smart code. During the refactoring process, the `MyClass` class may be rewritten in the following way:

```
class MyClass {
public:
    CObj *Find(const char *name);
    ...
} Storage;
```

After this modernization of the class, you should fix all the places in the program which use the `Find()` function. You cannot miss the first code fragment since it will not be compiled, so it will be certainly fixed:

```
CObj *obj = Storage.Find("ZZ");
if (obj != nullptr)
    Foo(obj);
```

The second code fragment is compiled well and you might miss it easily and therefore make the error we are discussing:

```
if (Storage.Find("foo") < 0)
    ObjectNotFound();
```

V504. It is highly probable that the semicolon ';' is missing after 'return' keyword.

The analyzer found a code fragment where the semicolon `';`' is probably missing. Here is an example of code that causes generating the V504 diagnostic message:

```
void Foo();

void Foo2(int *ptr)
{
    if (ptr == NULL)
        return
        Foo();
    ...
}
```

The programmer intended to terminate the function's operation if the pointer `ptr == NULL`. But the programmer forgot to write the semicolon `';`' after the `return` operator which causes the call of the `Foo()` function. The functions `Foo()` and `Foo2()` do not return anything and therefore the code is compiled without errors and warnings.

Most probably, the programmer intended to write:

```
void Foo();
```

```
void Foo2(int *ptr)
{
    if (ptr == NULL)
        return;
    Foo();
    ...
}
```

But if the initial code is still correct, it is better to rewrite it in the following way:

```
void Foo2(int *ptr)
{
    if (ptr == NULL)
    {
        Foo();
        return;
    }
    ...
}
```

The analyzer considers the code safe if the "if" operator is absent or the function call is located in the same line with the "return" operator. You might quite often see such code in programs. Here are examples of safe code:

```
void CPagerCtrl::RecalcSize()
{
    return
        (void)::SendMessageW(m_hWnd, (0x1400 + 2), 0, 0);
}

void Trace(unsigned int n, std::string const &s)
{ if (n) return TraceImpl(n, s); Trace0(s); }
```

V505. The 'alloca' function is used inside the loop. This can quickly overflow stack.

The analyzer detected a use of the `alloca` function inside a loop. Since the `alloca` function uses stack memory, its repeated call in the loop body might unexpectedly cause a stack overflow.

Here is an example of dangerous code:

```
for (size_t i = 0; i < n; ++i)
    if (wcscmp(strings[i], A2W(pszSrc[i])) == 0)
    {
        ...
    }
```

The `_alloca` function is used inside the `A2W` macro. Whether this code will cause an error or not depends upon the length of the processed strings, their number and size of the available stack.

V506. Pointer to local variable 'X' is stored outside the scope of this variable. Such a pointer will become invalid.

The analyzer found a potential error related to storing a pointer of a local variable. The warning is generated if the lifetime of an object is less than that of the pointer referring to it.

The first example:

```
class MyClass
{
    size_t *m_p;
    void Foo() {
        size_t localVar;
        ...
        m_p = &localVar;
    }
};
```

In this case, the address of the local variable is saved inside the class into the `m_p` variable and can be then used by mistake in a different function when the `localVar` variable is destructed.

The second example:

```
void Get(float **x)
{
    float f;
    ...
    *x = &f;
}
```

The `Get()` function will return the pointer to the local variable that will not exist by the moment.

This message is similar to [V507](#) message.

V507. Pointer to local array 'X' is stored outside the scope of this array. Such a pointer will become invalid.

The analyzer found a potential error related to storing a pointer of a local array. The warning is generated if the lifetime of an array is less than that of the pointer referring to it.

The first example:

```
class MyClass1
{
    int *m_p;
    void Foo()
    {
        int localArray[33];
        ...
    }
}
```

```

    m_x = localArray;
}
};

```

The localArray array is created in the stack and the localArray array will no longer exist after the Foo() function terminates. However, the pointer to this array will be saved in the m_p variable and can be used by mistake, which will cause an error.

The second example:

```

struct CVariable {
    ...
    char name[64];
};

void CRendererContext::RiGeometryV(int n, char *tokens[])
{
    for (i=0;i<n;i++)
    {
        CVariable var;
        if (parseVariable(&var, NULL, tokens[i])) {
            tokens[i] = var.name;
        }
    }
}

```

In this example, the pointer to the array situated in a variable of the CVariable type is saved in an external array. As a result, the "tokens" array will contain pointers to non-existing objects after the function RiGeometryV terminates.

The V507 warning does not always indicate an error. Below is an abridged code fragment that the analyzer considers dangerous although this code is correct:

```

png_infp info_ptr = png_create_info_struct(png_ptr);
...
BYTE trans[256];
info_ptr->trans = trans;
...
png_destroy_write_struct(&png_ptr, &info_ptr);

```

In this code, the lifetime of the info_ptr object coincides with the lifetime of trans. The object is created inside png_create_info_struct () and destroyed inside png_destroy_write_struct(). The analyzer cannot make out this case and supposes that the png_ptr object comes from outside. Here is an example where the analyzer could be right:

```

void Foo()
{
    png_infp info_ptr;
    info_ptr = GetExternInfoPng();
    BYTE trans[256];
    info_ptr->trans = trans;
}

```

This message is similar to [V506](#) message.

V508. The use of 'new type(n)' pattern was detected. Probably meant: 'new type[n]'.

The analyzer found code that might contain a misprint and therefore lead to an error. There is only one object of integer type that is dynamically created and initialized. It is highly probable that round brackets are used instead of square brackets by misprint. Here is an example:

```

int n;
...
int *P1 = new int(n);

```

Memory is allocated for one object of the int type. It is rather strange. Perhaps the correct code should look like this:

```

int n;
...
int *P1 = new int[n];

```

The analyzer generates the warning only if memory is allocated for simple types. The argument in the brackets must be of integer type in this case. As a result, the analyzer will not generate the warning on the following correct code:

```

float f = 1.0f;
float *f2 = new float(f);

MyClass *p = new MyClass(33);

```

V509. The 'throw' operator inside the destructor should be placed within the try..catch block. Raising exception inside the destructor is illegal.

In case an exception is thrown in a C++ program stack unwinding begins which causes objects to be destroyed by calling their destructors. If a destructor invoked during stack unwinding throws another exception and that exception propagates outside the destructor the C++ runtime immediately terminates the program by calling terminate() function. Therefore destructors should never let exceptions propagate - each exception thrown within a destructor should be handled in that destructor.

The analyzer found a destructor containing the throw operator outside the try..catch block. Here is an example:

```

LocalStorage::~LocalStorage()
{
    ...
    if (!FooFree(m_index))
        throw Err("FooFree", GetLastError());
    ...
}

```

This code must be rewritten so that the programmer is informed about the error that has occurred in the destructor without using the exception mechanism. If the error is not crucial, it can be ignored:

```

LocalStorage::~LocalStorage()
{
    try {
        ...
        if (!FooFree(m_index))
            throw Err("FooFree", GetLastError());
    }
}

```

```

    ...
}
catch (...)
{
    assert(false);
}
}

```

Additional materials on this topic:

1. Bjarne Stroustrup's C++ Style and Technique FAQ. Can I throw an exception from a constructor? From a destructor? <http://www.viva64.com/go.php?url=383>
2. Throwing destructors. <http://www.viva64.com/go.php?url=384>

V510. The 'Foo' function is not expected to receive class-type variable as 'N' actual argument.

[Note one specific thing about using the CString class from the MFC library](#)

[Related materials](#)

There are functions in whose description it is impossible to specify the number and types of all the acceptable parameters. In this case, the list of formal arguments ends with the ellipsis (...) that means: "and perhaps some more arguments". Here is an example of an ellipsis function: "int printf(const char* ...);". Only POD-types can serve as actual arguments for ellipsis.

POD is an abbreviation for "Plain Old Data", i.e. "Plain data in C style". The following types and structures refer to POD-types:

1. all the built-in arithmetic types (including wchar_t and bool);
2. types defined with the enum key word;
3. pointers;
4. POD-structures (struct or class) and POD-unions that meet the following requirements:
 - a. do not contain user constructors, destructor or copying assignment operator;
 - b. do not have base classes;
 - c. do not contain virtual functions;
 - d. do not contain protected or private non-static data members;
 - e. do not contain non-static data members of non-POD-types (or arrays of such types) and references.

If a class object is passed to an ellipsis function, this almost always indicates an error in program. The V510 rule helps detect incorrect code of the following kind:

```

wchar_t buf[100];
std::wstring ws(L"12345");
swprintf(buf, L"%s", ws);

```

The object's contents are saved into the stack instead of the pointer to the string. This code will cause generating "abracadabra" in the buffer or a program crash.

The correct version of the code must look this way:

```

wchar_t buf[100];
std::wstring ws(L"12345");
swprintf(buf, L"%s", ws.c_str());

```

Since you might pass anything you like into functions with a variable number of arguments, almost all the books on C++ programming do not recommend using them. They suggest employing safe mechanisms instead, for instance, boost::format.

Note one specific thing about using the CString class from the MFC library

We must see an error similar to the one mentioned above in the following code:

```

CString s;
CString arg(L"OK");
s.Format(L"Test CString: %s\n", arg);

```

The correct version of the code must look in the following way:

```

s.Format(L"Test CString: %s\n", arg.GetString());

```

Or, as MSDN suggests [1], you may use the explicit cast operator LPCTSTR implemented in the CString class to get a pointer to the string. Here is a sample of correct code from MSDN:

```

CString kindOfFruit = "bananas";
int howmany = 25;
printf("You have %d %s\n", howmany, (LPCTSTR)kindOfFruit);

```

However, the first version "s.Format(L"Test CString: %s\n", arg);" is actually correct as well like the others. This topic is discussed in detail in the article "[Big Brother helps you](#)" [2].

The MFC developers implemented the CString class in a special way so that you could pass it into functions of the printf and Format types. It is done rather intricately and if you want to make it out, study implementation of the CStringT class in the source codes and also read a detailed discussion "[Pass CString to printf?](#)" [3].

So, the analyzer makes an exception for the CStringT type and considers the following code correct:

```

CString s;
CString arg(L"OK");
s.Format(L"Test CString: %s\n", arg);

```

Related materials

1. MSDN. CString Operations Relating to C-Style Strings. <http://www.viva64.com/go.php?url=393>
2. OOO "Program Verification Systems" blog. Big Brother helps you. <http://www.viva64.com/en/b/0073/>
3. Discussion at eggheadcafe.com. Pass CString to printf? <http://www.viva64.com/go.php?url=394>

V511. The sizeof() operator returns size of the pointer, and not of the array, in given expression.

There is one specific feature of the language you might easily forget about and make a mistake. Look at the following code fragment:

```
char A[100];
void Foo(char B[100])
{
}
```

In this code, the A object is an array and the sizeof(A) expression will return value 100.

The B object is simply a pointer. Value 100 in the square brackets indicates to the programmer that he is working with an array of 100 items. But it is not an array of a hundred items which is passed into the function - it is only the pointer. So, the sizeof(B) expression will return value 4 or 8 (the size of the pointer in a 32-bit/64-bit system).

The V511 warning is generated when the size of a pointer is calculated which is passed as an argument in the format "TypeName ArrayName[N]". Such code is most likely to have an error. Look at the sample:

```
void Foo(float array[3])
{
    size_t n = sizeof(array) / sizeof(array[0]);
    for (size_t i = 0; i != n; i++)
        array[i] = 1.0f;
}
```

The function will not fill the whole array with value 1.0f but only 1 or 2 items depending on the system's capacity.

Win32: sizeof(array) / sizeof(array[0]) = 4/4 = 1.

Win64: sizeof(array) / sizeof(array[0]) = 8/4 = 2.

To avoid such errors, we must explicitly pass the array's size. Here is correct code:

```
void Foo(float *array, size_t arraySize)
{
    for (size_t i = 0; i != arraySize; i++)
        array[i] = 1.0f;
}
```

Another way is to use a reference to the array:

```
void Foo(float (&array)[3])
{
    size_t n = sizeof(array) / sizeof(array[0]);
    for (size_t i = 0; i != n; i++)
        array[i] = 1.0f;
}
```

V512. A call of the 'Foo' function will lead to a buffer overflow or underflow.

The analyzer found a potential error related to memory buffer filling, copying or comparison. The error might cause a buffer overflow or, vice versa, buffer underflow.

This is a rather common kind of errors that occurs due to misprints or inattention. What is unpleasant about such errors is that a program might work well for a long time. Due to sheer luck, acceptable values might be found in uninitialized memory. The area of writable memory might not be used.

Let's study two samples taken from real applications.

Sample N1.

```
MD5Context *ctx;
...
memset(ctx, 0, sizeof(ctx));
```

Here the misprint causes release of only a part of the structure and not the whole structure. The error is in calculation of the pointer's size and not the whole structure MD5Context. Here is the correct version of the code:

```
MD5Context *ctx;
...
memset(ctx, 0, sizeof(*ctx));
```

Sample N2.

```
#define CONT_MAP_MAX 50
int _iContMap[CONT_MAP_MAX];
memset(_iContMap, -1, CONT_MAP_MAX);
```

In this sample, the size of the buffer to be filled is also defined incorrectly. This is the correct version:

```
#define CONT_MAP_MAX 50
int _iContMap[CONT_MAP_MAX];
memset(_iContMap, -1, CONT_MAP_MAX * sizeof(int));
```

V513. Use _beginthreadex/_endthreadex functions instead of CreateThread/ExitThread functions.

A use of the CreateThread function or ExitThread function is detected in a program. If CRT (C run-time library) functions are used in concurrent threads, you should call the functions _beginthreadex/_endthreadex instead of CreateThread/ExitThread.

Below is an extract from the 6-th chapter of the book "Advanced Windows: creating efficient Win32-applications considering the specifics of the 64-bit Windows" by Jeffrey Richter / 4-th issue.

CreateThread is a Windows-function creating a thread. But never call it if you write your code in C/C++. You should use the function _beginthreadex from the Visual C++ library instead.

To make multi-threaded applications using C/C++ (CRT) library work correctly, you should create a special data structure and link it to every thread from which the library functions are called. Moreover, they must know that when you address them, they must look through this data block in the master thread in order not to damage data in some other thread.

So how does the system know that it must create this data block together with creating a new thread? The answer is very simple - it doesn't know and never will like to. Only you are fully responsible for it. If you use functions which are unsafe in multi-threaded environment, you should create threads with the library

function `_beginthreadex` and not Windows-function `CreateThread`.

Note that the `_beginthreadex` function exists only in multi-threaded versions of the C/C++ library. When linking a project to a single-threaded library, the linker will generate an error message "unresolved external symbol". Of course, it is done intentionally since the single-threaded library cannot work correctly in a multi-threaded application. Note also that Visual Studio chooses the single-threaded library by default when creating a new project. This way is not the safest one, so you should choose yourself one of the multi-threaded versions of the C/C++ library for multi-threaded applications.

Correspondingly, you must use the function `_endthreadex` to destruct a thread created with the function `_beginthreadex`.

Additional materials on this topic:

1. Discussion at StackOverflow. "Windows threading: `_beginthread` vs `_beginthreadex` vs `CreateThread` C++". <http://www.viva64.com/go.php?url=385>
2. Discussion at CodeGuru Forum. " `_beginthread` vs `CreateThread`". <http://www.viva64.com/go.php?url=386>
3. Discussion at MSDN forum. "CreateThread vs `_beginthreadex`". <http://www.viva64.com/go.php?url=387>
4. Description of using C Run-Time (CRT) functions and `CreateThread()`. <http://www.viva64.com/go.php?url=407>

V514. Dividing `sizeof` of a pointer by another value. There is a probability of logical error presence.

The analyzer found a potential error related to division of the pointer's size by some value. Division of the pointer's size is rather a strange operation since it has no practical sense and most likely indicates an error or misprint in code.

Consider an example:

```
const size_t StrLen = 16;
LPTSTR dest = new TCHAR[StrLen];
TCHAR src[StrLen] = _T("PVS-Studio V514");
_tcsncpy(dest, src, sizeof(dest)/sizeof(dest[0]));
```

In the "`sizeof(dest)/sizeof(dest[0])`" expression, the pointer's size is divided by the size of the element the pointer refers to. As a result, we might get different numbers of copied bytes depending on sizes of the pointer and `TCHAR` type - but never the number the programmer expected.

Taking into account that the `_tcsncpy` function is unsafe in itself, correct and safer code may look in the following way:

```
const size_t StrLen = 16;
LPTSTR dest = new TCHAR[StrLen];
TCHAR src[StrLen] = _T("PVS-Studio V514");
_tcsncpy_s(dest, StrLen, src, StrLen);
```

V515. The 'delete' operator is applied to non-pointer.

In code, the delete operator is applied to a class object instead of the pointer. It is most likely to be an error.

Consider a code sample:

```
CString str;
...
delete str;
```

The 'delete' operator can be applied to an object of the `CString` type since the `CString` class can be automatically cast to the pointer. Such code might cause an exception or unexpected program behavior.

Correct code might look so:

```
CString *pstr = new CString;
...
delete pstr;
```

In some cases, applying the 'delete' operator to class objects is not an error. You may encounter such code, for instance, when working with the `QT::QBasicAtomicPointer` class. The analyzer ignores calls of the 'delete' operator to objects of this type. If you know other similar classes it is a normal practice to apply the 'delete' operator to, please tell us about them. We will add them into exceptions.

V516. Consider inspecting an odd expression. Non-null function pointer is compared to null.

Code contains a construct comparing a non-null pointer to a function with null. It is most probably that there is a misprint in code – parentheses are missing.

Consider this example:

```
int Foo();
void Use()
{
    if (Foo == 0)
    {
        //...
    }
}
```

The condition "`Foo == 0`" is meaningless. The address of the 'Foo' function never equals zero, so the comparison result will always be 'false'. In the code we consider, the programmer missed parentheses by accident. This is the correct version of the code:

```
if (Foo() == 0)
{
    //...
}
```

If there is an explicit taking of address, the code is considered correct. For example:

```
int Foo();
void Use()
{
    if (&Foo != NULL)
        //...
}
```

V517. The use of 'if (A) {...} else if (A) {...}' pattern was detected. There is

a probability of logical error presence.

The analyzer detected a possible error in a construct consisting of conditional statements. Consider the sample:

```
if (a == 1)
    Foo1();
else if (a == 2)
    Foo2();
else if (a == 1)
    Foo3();
```

In this sample, the 'Foo3()' function will never get control. Most likely, we deal with a logical error and the correct code should look as follows:

```
if (a == 1)
    Foo1();
else if (a == 2)
    Foo2();
else if (a == 3)
    Foo3();
```

In practice, such an error might look in the following way:

```
If (radius < THRESH * 5)
    *yOut = THRESH * 10 / radius;
else if (radius < THRESH * 5)
    *yOut = -3.0f / (THRESH * 5.0f) * (radius - THRESH * 5.0f) + 3.0f;
else
    *yOut = 0.0f;
```

It is difficult to say how a correct comparison condition must look, but the error in this code is evident.

V518. The 'malloc' function allocates strange amount of memory calculated by 'strlen(expr)'. Perhaps the correct variant is strlen(expr) + 1.

The analyzer found a potential error related to allocating insufficient amount of memory. The string's length is calculated in code and the memory buffer of a corresponding size is allocated but the terminal '\0' is not allowed for.

Consider this example:

```
char *p = (char *)malloc(strlen(src));
strcpy(p, src);
```

In this case, it is just +1 which is missing. The correct version is:

```
char *p = (char *)malloc(strlen(src) + 1);
strcpy(p, src);
```

Here is another example of incorrect code detected by the analyzer in one application:

```
if((t=(char *)realloc(next->name, strlen(name+1))))
{
    next->name=t;
    strcpy(next->name,name);
}
```

The programmer was inattentive and made a mistake when writing the right bracket ')'. As a result, we will allocate 2 bytes less memory than necessary. This is the correct code:

```
if((t=(char *)realloc(next->name, strlen(name)+1)))
```

V519. The 'x' variable is assigned values twice successively. Perhaps this is a mistake.

The analyzer detected a potential error related to assignment of a value two times successively to the same variable while the variable itself is not used between these assignments.

Consider this sample:

```
A = GetA();
A = GetB();
```

The fact that the 'A' variable is assigned values twice might signal an error. Most probably, the code should look this way:

```
A = GetA();
B = GetB();
```

If the variable is used between assignments, the analyzer considers this code correct:

```
A = 1;
A = A + 1;
A = Foo(A);
```

Let's see how such an error may look in practice. The following sample is taken from a real application where a user class CSize is implemented:

```
class CSize : public SIZE
{
    ...
    CSize(POINT pt) { cx = pt.x; cy = pt.y; }
```

The correct version is the following:

```
CSize(POINT pt) { cx = pt.x; cy = pt.y; }
```

Let's study one more example. The second line was written for the purpose of debugging or checking how text of a different color would look. And it seems that the programmer forgot to remove the second line then:

```
m_clrSample = GetSysColor(COLOR_WINDOWTEXT);
```

```
m_clrSample = RGB(60,0,0);
```

Sometimes the analyzer generates false alarms when writing into variables is used for the purpose of debugging. Here is an example of such code:

```
status = Foo1();
status = Foo2();
status = Foo3();
```

In this case, we may suppress false alarms using the "//V519" comment. We may also remove meaningless assignments from the code. And the last thing. Perhaps this code is still incorrect, so we have to check the value of the 'status' variable.

V520. The comma operator ',' in array index expression.

The analyzer found a potential error that may be caused by a misprint. An expression containing the ',' operator is used as an index for an array.

Here is a sample of suspicious code:

```
float **array_2D;
array_2D[getx() , gety()] = 0;
```

Most probably, it was meant to be:

```
array_2D[ getx() ][ gety() ] = 0;
```

Such errors might appear if the programmer worked earlier with a programming language where array indexes are separated by commas.

Let's look at a sample of an error found by the analyzer in one project:

```
float **m;
TextOutput &t = ...
...
t.printf("%10.5f, %10.5f, %10.5f,\n%10.5f, %10.5f, %10.5f,\n%10.5f, %10.5f, %10.5f)",
    m[0][0], m[0][1], m[0][2],
    m[1][0], m[1][1], m[1][2],
    m[2][0], m[2][1], m[2][2]);
```

Since the printf function of the TextOutput class works with a variable number of arguments, it cannot check whether pointers will be passed to it instead of values of the float type. As a result, we will get rubbish displayed instead of matrix items' values. This is the correct code:

```
t.printf("%10.5f, %10.5f, %10.5f,\n%10.5f, %10.5f, %10.5f,\n%10.5f, %10.5f, %10.5f)",
    m[0][0], m[0][1], m[0][2],
    m[1][0], m[1][1], m[1][2],
    m[2][0], m[2][1], m[2][2]);
```

V521. Such expressions using the ',' operator are dangerous. Make sure the expression is correct.

The comma operator ',' is used to execute expressions to the both sides of it in the left-to-right order and get the value of the right expression.

The analyzer found an expression in code that uses the ',' operator in a suspicious way. It is highly probable that the program text contains a misprint.

Consider the following sample:

```
float Foo()
{
    double A;
    A = 1,23;
    float f = 10.0f;
    return 3,f;
}
```

In this code, the A variable will be assigned value 1 instead of 1.23. According to C/C++ rules, the "A = 1,23" expression equals "(A = 1),23". Also, the Foo() function will return value 10.0f instead of 3.0f. In the both cases, the error is related to using the ',' character instead of the '.' character.

This is the corrected version:

```
float Foo()
{
    double A;
    A = 1.23;
    float f = 10.0f;
    return 3.f;
}
```

Note. There were cases when the analyzer could not make out the code and generated V521 warnings for absolutely safe constructs. It is usually related to usage of template classes or complex macros. If you noted such a false alarm when working with the analyzer, please tell the developers about it. To suppress false alarms, you may use the comment of the "//V521" type.

V522. Dereferencing of the null pointer might take place. Check the logical condition.

The analyzer detected a fragment of code that might cause using a null pointer.

Let's study several examples the analyzer generates the V522 diagnostic message for:

```
if (pointer != 0 || pointer->m_a) { ... }
if (pointer == 0 && pointer->x()) { ... }
if (array == 0 && array[3]) { ... }
if (!pointer && pointer->x()) { ... }
```

In all the conditions, there is a logical error that leads to dereferencing of the null pointer. The error may be introduced into the code during code refactoring or through a misprint.

Correct versions:

```
if (pointer != 0 && pointer->m_a) { ... }
if (pointer != 0 && pointer->x()) { ... }
if (array != 0 && array[3]) { ... }
```

```
if (pointer && pointer->x()) { ... }
```

V523. The 'then' statement is equivalent to the 'else' statement.

The analyzer found a case when the true and false statements of the 'if' operator coincide completely. This often signals a logical error.

Here is an example:

```
if (X)
    Foo_A();
else
    Foo_A();
```

Whether the X condition is false or true, the Foo_A() function will be called anyway.

This is the correct version of the code:

```
if (X)
    Foo_A();
else
    Foo_B();
```

Here is an example of such an error taken from a real application:

```
if (!_isVertical)
    Flags |= DT_BOTTOM;
else
    Flags |= DT_BOTTOM;
```

Presence of two empty statements is considered correct and safe. You might often see such constructs when using macros. This is a sample of safe code:

```
if (exp) {
} else {
}
```

V524. It is odd that the body of 'Foo_1' function is fully equivalent to the body of 'Foo_2' function.

This warning is generated when the analyzer detects two functions implemented in the same way. Presence of two identical functions is not an error in itself but you should study them.

The sense of such diagnosis is detecting the following type of errors:

```
class Point
{
    ...
    float GetX() { return m_x; }
    float GetY() { return m_x; }
};
```

The misprint in the code causes the two functions different in sense to perform the same actions. This is the correct version:

```
float GetX() { return m_x; }
float GetY() { return m_y; }
```

Identity of the bodies of functions GetX() and GetY() in this sample obviously signals an error. However, the percentage of false alarms will be too great if the analyzer generates warnings for all identical functions, so it is guided by a range of exceptions when it must not warn the programmer about identical function bodies. Here are some of them:

- The analyzer does not report about identity of functions' bodies if they do not use variables except for arguments. For example: "bool IsXYZ() { return true; }".
- Functions use static objects and therefore have different inner states. For example: "int Get() { static int x = 1; return x++; }"
- Functions are type cast operators.
- Functions with identical bodies are repeated more than twice.
- And so on.

However, in some cases the analyzer cannot understand that identical function bodies are not an error. This is code which is diagnosed as dangerous but really it is not:

```
PolynomialMod2 Plus(const PolynomialMod2 &b) const {return Xor(b);}
PolynomialMod2 Minus(const PolynomialMod2 &b) const {return Xor(b);}
```

You can suppress false alarms using several methods. If false alarms refer to files of external libraries, you may add this library (i.e. its path) to exceptions. If false alarms refer to your own code, you may use the comment of the "//V524" type to suppress false warnings. If there are too many false alarms, you may completely disable this diagnosis in the analyzer's settings. You may also modify the code so that one function calls another with the same code.

The last method is often the best since it, first, reduces the amount of code and, second, makes it easier to support. You need to edit only one function instead of the both functions. This is a sample of real code where the programmer could benefit from calling one function from another:

```
static void PreSave(void) {
    int x;
    for(x=0;x<TotalSides;x++) {
        int b;
        for(b=0; b<65500; b++)
            diskdata[x][b] ^= diskdatao[x][b];
    }
}

static void PostSave (void) {
    int x;
    for(x=0;x<TotalSides;x++) {
        int b;
        for(b=0; b<65500; b++)
            diskdata[x][b] ^= diskdatao[x][b];
    }
}
```

This code should be replaced with the following:

```
static void PreSave(void) {
    int x;
    for(x=0;x<TotalSides;x++) {
        int b;
```

```

        for(b=0; b<65500; b++)
            diskdata[x][b] ^= diskdatao[x][b];
    }
}

static void PostSave (void) {
    PreSave();
}

```

We did not fix the error in this sample, but the V524 warning disappeared after refactoring and the code got simpler.

V525. The code containing the collection of similar blocks. Check items X, Y, Z, ... in lines N1, N2, N3, ...

The analyzer detected code that might contain a misprint. This code can be split into smaller similar fragments. Although they look similar, they differ in some way. It is highly probable that this code was created with the Copy-Paste method. The V525 message is generated if the analyzer suspects that some element was not fixed in the copied text. The error might be located in one of the lines whose numbers are listed in the V525 message.

Disadvantages of the V525 message:

- 1) This diagnostic rule is based on heuristic methods and often produces false alarms.
- 2) Implementation of the rule's heuristic algorithm is complicated and occupies more than 1000 lines of C++ code. That is why it is difficult to describe in documentation. So it may be hard for the user to understand why the V525 message was generated.
- 3) The diagnostic message refers not to one line but several lines. The analyzer cannot point out only one line since the error may be in any of them.

Advantages of the V525 message:

- 1) It can detect errors which are too hard to notice during code review.

Let's study an artificial sample at first:

```

...
float rgba[4];
rgba[0] = object.GetR();
rgba[1] = object.GetG();
rgba[2] = object.GetB();
rgba[3] = object.GetR();

```

The 'rgba' array presents color and transparency of some object. When writing the code that fills the array, we wrote the line "rgba[0] = object.GetR();" at first. Then we copied and changed this line several times. But in the last line, we missed some changes, so it is the 'GetR()' function which is called instead of the 'GetA()' function. The analyzer generates the following warning on this code:

V525: The code containing the collection of similar blocks. Check items 'GetR', 'GetG', 'GetB', 'GetR' in lines 12, 13, 14, 15.

If you review lines 12, 13, 14 and 15, you will find the error. This is the correct code:

```

rgba[3] = object.GetA();

```

Now let's study several samples taken from real applications. The first sample:

```

tbb[0].iBitmap = 0;
tbb[0].idCommand = IDC_TB_EXIT;
tbb[0].fsState = TBSTATE_ENABLED;
tbb[0].fsStyle = BTNS_BUTTON;
tbb[0].dwData = 0;
tbb[0].iString = -1;
...
tbb[6].iBitmap = 6;
tbb[6].idCommand = IDC_TB_SETTINGS;
tbb[6].fsState = TBSTATE_ENABLED;
tbb[6].fsStyle = BTNS_BUTTON;
tbb[6].dwData = 0;
tbb[6].iString = -1;

tbb[7].iBitmap = 7;
tbb[7].idCommand = IDC_TB_CALC;
tbb[7].fsState = TBSTATE_ENABLED;
tbb[7].fsStyle = BTNS_BUTTON;
tbb[6].dwData = 0;
tbb[7].iString = -1;

```

The code fragment is far not complete. More than half of it was cut out. The fragment was being written through copying and editing the code. No wonder that an incorrect index was lost in such a large fragment. The analyzer generates the following diagnostic message: "The code containing the collection of similar blocks. Check items '0', '1', '2', '3', '4', '5', '6', '6' in lines 589, 596, 603, 610, 617, 624, 631, 638". If we review these lines, we will find and correct the index '6' repeated twice. This is the correct code:

```

tbb[7].iBitmap = 7;
tbb[7].idCommand = IDC_TB_CALC;
tbb[7].fsState = TBSTATE_ENABLED;
tbb[7].fsStyle = BTNS_BUTTON;
tbb[7].dwData = 0;
tbb[7].iString = -1;

```

The second sample:

```

pPopup->EnableMenuItem(
    ID_CONTEXT_EDITTEXT, MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
pPopup->EnableMenuItem(
    ID_CONTEXT_CLOSEALL, MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
pPopup->EnableMenuItem(
    ID_CONTEXT_CLOSE, MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
pPopup->EnableMenuItem(
    ID_CONTEXT_SAVELAYOUT, MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
pPopup->EnableMenuItem(
    ID_CONTEXT_RESIZE, MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
pPopup->EnableMenuItem(
    ID_CONTEXT_REFRESH, MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
pPopup->EnableMenuItem(
    ID_CONTEXT_EDITTEXT, MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
pPopup->EnableMenuItem(
    ID_CONTEXT_SAVE, MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
pPopup->EnableMenuItem(

```

```
ID_CONTEXT_EDITIMAGE,MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
pPopup->EnableMenuItem(
    ID_CONTEXT_CLONE,MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
```

It is very difficult to find an error in this code while reviewing it. But there *is* an error here: the state of the same menu item 'ID_CONTEXT_EDITTEXT' is modified twice. Let's mark the two repeated lines:

```
-----
pPopup->EnableMenuItem(
    ID_CONTEXT_EDITTEXT,MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
-----
pPopup->EnableMenuItem(
    ID_CONTEXT_CLOSEALL, MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
pPopup->EnableMenuItem(
    ID_CONTEXT_CLOSE, MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
pPopup->EnableMenuItem(
    ID_CONTEXT_SAVELAYOUT, MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
pPopup->EnableMenuItem(
    ID_CONTEXT_RESIZE, MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
pPopup->EnableMenuItem(
    ID_CONTEXT_REFRESH, MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
-----
pPopup->EnableMenuItem(
    ID_CONTEXT_EDITTEXT, MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
-----
pPopup->EnableMenuItem(
    ID_CONTEXT_SAVE, MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
pPopup->EnableMenuItem(
    ID_CONTEXT_EDITIMAGE,MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
pPopup->EnableMenuItem(
    ID_CONTEXT_CLONE,MF_GRAYED|MF_DISABLED|MF_BYCOMMAND);
```

Maybe it is a small error and one of the lines is just unnecessary. Or maybe the programmer forgot to change the state of some other menu item.

Unfortunately, the analyzer often makes a mistake while carrying out this diagnosis and generates false alarms. This is an example of code causing a false alarm:

```
switch (i) {
    case 0: f1 = 2; f2 = 3; break;
    case 1: f1 = 0; f2 = 3; break;
    case 2: f1 = 1; f2 = 3; break;
    case 3: f1 = 1; f2 = 2; break;
    case 4: f1 = 2; f2 = 0; break;
    case 5: f1 = 0; f2 = 1; break;
}
```

The analyzer does not like a correct column of numbers: 2, 0, 1, 1, 2, 0. In such cases, you may enable the warning suppression mechanism by typing the comment `//V525` in the end of the line:

```
switch (i) {
    case 0: f1 = 2; f2 = 3; break; //V525
    case 1: f1 = 0; f2 = 3; break;
    case 2: f1 = 1; f2 = 3; break;
    case 3: f1 = 1; f2 = 2; break;
    case 4: f1 = 2; f2 = 0; break;
    case 5: f1 = 0; f2 = 1; break;
}
```

If there are too many false alarms, you may disable this diagnostic rule in the analyzer's settings. We will also appreciate if you write to our support service about cases when false alarms are generated and we will try to improve the diagnosis algorithm. Please attach corresponding code fragments to your letters.

V526. The 'strcmp' function returns 0 if corresponding strings are equal. Consider examining the condition for mistakes.

This message is a kind of recommendation. It rarely diagnoses a logical error but helps make code more readable for young developers.

The analyzer detected a construct comparing two strings that can be written in a clearer way. Such functions as `strcmp`, `strncmp` and `wcsncmp` return 0 if strings identical. It may cause logical errors in program. Look at a code sample:

```
if (strcmp(s1, s2))
```

This condition will hold if the strings ARE NOT IDENTICAL. Perhaps you remember well what `strcmp()` returns, but a person who rarely works with string functions might think that the `strcmp()` function returns the value of type 'bool'. Then he will read this code in this way: "the condition is true if the strings match".

You'd better not save on more characters in the program text and write the code this way:

```
if (strcmp(s1, s2) != 0)
```

This text tells the programmer that the `strcmp()` function returns some numeric value, not the bool type. This code ensures that the programmer will understand it properly.

If you do not want to get this diagnostic message, you may disable it in the PVS-Studio's settings.

V527. It is odd that the 'zero' value is assigned to pointer. Probably meant: *ptr = zero.

This error occurs in two similar cases.

1) The analyzer found a potential error: a pointer to bool type is assigned false value. It is highly probable that the pointer dereferencing operation is missing. For example:

```
float Get(bool *retStatus)
{
    ...
    if (retStatus != nullptr)
        retStatus = false;
    ...
}
```

The `""` operator is missing in this code. The operation of nulling the `retStatus` pointer will be performed instead of status return. This is the correct code:

```
if (retStatus != nullptr)
```

```
*retStatus = false;
```

2) The analyzer found a potential error: a pointer referring to the char/wchar_t type is assigned value '0' or L'0'. It is highly probable that the pointer dereferencing operation is missing. For example:

```
char *cp;
...
cp = '\0';
```

This is the correct code:

```
char *cp;
...
*cp = '\0';
```

V528. It is odd that pointer is compared with the 'zero' value. Probably meant: *ptr != zero.

This error occurs in two similar cases.

1) The analyzer found a potential error: a pointer to bool type is compared to false value. It is highly probable that the pointer dereferencing operation is missing. For example:

```
bool *pState;
...
if (pState != false)
...
```

The '*' operator is missing in this code. As a result, we compare the pState pointer's value to the null pointer. This is the correct code:

```
bool *pState;
...
if (*pState != false)
...
```

2) The analyzer found a potential error: a pointer to the char/wchar_t type is compared to value '0' or L'0'. It is highly probable that the pointer dereferencing operation is missing. For example:

```
char *cp;
...
if (cp != '\0')
```

This is the correct code:

```
char *cp;
...
if (*cp != '\0')
```

V529. Odd semicolon ';' after 'if/for/while' operator.

The analyzer detected a potential error: a semicolon ';' stands after the 'if', 'for' or 'while' operator. For example:

```
for (i = 0; i < n; i++);
{
    Foo(i);
}
```

This is the correct code:

```
for (i = 0; i < n; i++)
{
    Foo(i);
}
```

Using a semicolon ';' right after the for or while operator is not an error in itself and you may see it quite often in code. So the analyzer eliminates many cases relying on some additional factors. For instance, the following code sample is considered safe:

```
for (depth = 0, cur = parent; cur; depth++, cur = cur->parent)
;
```

V530. The return value of function 'Foo' is required to be utilized.

Calls of some functions are senseless if their results are not used. Let's study the first sample:

```
void VariantValue::Clear()
{
    m_vtype = VT_NULL;
    m_bvalue = false;
    m_ivalue = 0;
    m_fvalue = 0;
    m_svalue.empty();
    m_tvalue = 0;
}
```

This value emptying code is taken from a real application. The error here is the following: by accident, the string::empty() function is called instead of the string::clear() function and the line's content remains unchanged. The analyzer diagnoses this error relying on knowledge that the result of the string::empty() function must be used. For instance, it must be compared to something or written into a variable.

This is the correct code:

```
void VariantValue::Clear()
{
    m_vtype = VT_NULL;
    m_bvalue = false;
    m_ivalue = 0;
    m_fvalue = 0;
    m_svalue.clear();
    m_tvalue = 0;
}
```

The second sample:


```
void unregisterThread() {
    Guard<TaskQueue> g(_taskQueue);
    std::remove(_threads.begin(), _threads.end(),
                ThreadImpl::current());
}
```

The `std::remove` function does not remove elements from the container. It only shifts the elements and brings the iterator back to the beginning of the trash. Suppose we have the `vector<int>` container that contains elements 1,2,3,1,2,3,1,2,3. If we execute the code "`remove(v.begin(), v.end(), 2)`", the container will contain elements 1,3,1,3,?,?, where ? is some trash. The function will bring the iterator back to the first senseless element, so if we want to remove these trash elements, we must write the code this way: "`v.erase(remove(v.begin(), v.end(), 2), v.end())`".

As you may see from this explanation, the result `std::remove` must be used. This is the correct code:

```
void unregisterThread() {
    Guard<TaskQueue> g(_taskQueue);
    auto trash = std::remove(_threads.begin(), _threads.end(),
                            ThreadImpl::current());
    _threads.erase(trash, _threads.end());
}
```

There are very many functions whose results must be used. Among them are the following: `malloc`, `realloc`, `fopen`, `isalpha`, `atof`, `strcmp` and many, many others. An unused result signals an error which is usually caused by a misprint. However, the PVS-Studio analyzer warns only about errors related to using the STL library. There are two reasons for that:

- 1) It is much more difficult to make a mistake by not using the result of the `fopen()` function than confuse `std::clear()` and `std::empty()`.
- 2) This functionality duplicates the capabilities of Code Analysis for C/C++ included into some Visual Studio editions (see warning C6031). But these warnings are not implemented in Visual Studio for STL functions.

If you want to propose extending the list of functions supported by PVS-Studio, contact our support service. We will appreciate if you give interesting samples and advice.

V531. It is odd that a `sizeof()` operator is multiplied by `sizeof()`.

Code where a value returned by the `sizeof()` operator is multiplied by another `sizeof()` operator most always signals an error. It is unreasonable to multiply the size of one object by the size of another object. Such errors usually occur when working with strings.

Let's study a real code sample:

```
TCHAR szTemp[256];
DWORD dwLen =
    ::LoadString(hInstDll, dwID, szTemp,
                sizeof(szTemp) * sizeof(TCHAR));
```

The `LoadString` function takes the buffer's size in characters as the last argument. In the Unicode version of the application, we will tell the function that the buffer's size is larger than it is actually. This may cause a buffer overflow. Note that if we fix the code in the following way, it will not become correct at all:

```
TCHAR szTemp[256];
DWORD dwLen =
    ::LoadString(hInstDll, dwID, szTemp, sizeof(szTemp));
```

Here is a quotation from MSDN on this topic:

Using this function incorrectly can compromise the security of your application. Incorrect use includes specifying the wrong size in the `nBufferMax` parameter. For example, if `lpBuffer` points to a buffer `szBuffer` which is declared as `TCHAR szBuffer[100]`, then `sizeof(szBuffer)` gives the size of the buffer in bytes, which could lead to a buffer overflow for the Unicode version of the function. Buffer overflow situations are the cause of many security problems in applications. In this case, using `sizeof(szBuffer)/sizeof(TCHAR)` or `sizeof(szBuffer)/sizeof(szBuffer[0])` would give the proper size of the buffer.

This is the correct code:

```
TCHAR szTemp[256];
DWORD dwLen =
    ::LoadString(hInstDll, dwID, szTemp,
                sizeof(szTemp) / sizeof(TCHAR));
```

Here is another correct code:

```
const size_t BUF_LEN = 256;
TCHAR szTemp[BUF_LEN];
DWORD dwLen =
    ::LoadString(hInstDll, dwID, szTemp, BUF_LEN);
```

V532. Consider inspecting the statement of '*pointer++' pattern. Probably meant: '(*pointer)++'.

The analyzer detected a potential error: a pointer dereferencing operation is present in code but the value the pointer refers to is not used in any way.

Let's study this sample:

```
int *p;
...
*p++;
```

The "`*p++`" expression performs the following actions. The "`p`" pointer is incremented by one, but before that a value of the "`int`" type is fetched from memory. This value is not used in any way, which is strange. It looks as if the dereferencing operation "`*`" is unnecessary. There are several ways of correcting the code:

- 1) We may remove the unnecessary dereferencing operation - the "`*p++;`" expression is equal to "`p++;`":

```
int *p;
...
p++;
```

- 2) If the developer intended to increment the value instead of the pointer, we should write it so:

```
int *p;
...
(*p)++;
```

If the "`*p++`" expression's result is used, the analyzer considers the code correct. This is a sample of safe code:

```
while(*src)
    *dest++ = *src++;
```

Let's study a sample taken from a real application:

```
STDMETHODIMP CCustomAutoComplete::Next(
    ULONG celt, LPOLESTR *rgelt, ULONG *pceltFetched)
{
    ...
    if (pceltFetched != NULL)
        *pceltFetched++;
    ...
}
```

In this case, parentheses are missing. This is the correct code:

```
if (pceltFetched != NULL)
    (*pceltFetched)++;
```

V533. It is likely that a wrong variable is being incremented inside the 'for' operator. Consider reviewing 'X'.

The analyzer detected a potential error: a variable referring to an outer loop and located inside the 'for' operator is incremented.

This is the simplest form of this error:

```
for (size_t i = 0; i != 5; i++)
    for (size_t j = 0; j != 5; i++)
        A[i][j] = 0;
```

It is the 'i' variable which is incremented instead of 'j' in the inner loop. Such an error might be not so visible in a real application. This is the correct code:

```
for (size_t i = 0; i != 5; i++)
    for (size_t j = 0; j != 5; j++)
        A[i][j] = 0;
```

V533. It is likely that a wrong variable is being incremented inside the 'for' operator. Consider reviewing 'X'.

The analyzer detected a potential error: a variable referring to an outer loop and located inside the 'for' operator is incremented.

This is the simplest form of this error:

```
for (size_t i = 0; i != 5; i++)
    for (size_t j = 0; j != 5; i++)
        A[i][j] = 0;
```

It is the 'i' variable which is incremented instead of 'j' in the inner loop. Such an error might be not so visible in a real application. This is the correct code:

```
for (size_t i = 0; i != 5; i++)
    for (size_t j = 0; j != 5; j++)
        A[i][j] = 0;
```

V534. It is likely that a wrong variable is being compared inside the 'for' operator. Consider reviewing 'X'.

The analyzer detected a potential error: a variable referring to an outer loop is used in the condition of the 'for' operator.

This is the simplest form of this error:

```
for (size_t i = 0; i != 5; i++)
    for (size_t j = 0; i != 5; j++)
        A[i][j] = 0;
```

It is the comparison 'i != 5' that is performed instead of 'j != 5' in the inner loop. Such an error might be not so visible in a real application. This is the correct code:

```
for (size_t i = 0; i != 5; i++)
    for (size_t j = 0; j != 5; j++)
        A[i][j] = 0;
```

V535. The variable 'X' is being used for this loop and for the outer loop.

The analyzer detected a potential error: a nested loop is arranged by a variable which is also used in an outer loop. In a schematic form, this error looks in the following way:

```
size_t i, j;
for (i = 0; i != 5; i++)
    for (i = 0; i != 5; i++)
        A[i][j] = 0;
```

Of course, this is an artificial sample, so we may easily see the error, but in a real application, the error might be not so apparent. This is the correct code:

```
size_t i, j;
for (i = 0; i != 5; i++)
    for (j = 0; j != 5; j++)
        A[i][j] = 0;
```

Sometimes the analyzer generates false alarms. Let's study a code sample causing the analyzer to produce a false alarm:

```
for(c = lb; c <= ub; c++)
{
    if (!(xlb <= xlat(c) && xlat(c) <= ub))
    {
        Range * r = new Range(xlb, xlb + 1);
        for (c = lb + 1; c <= ub; c++)
            r = doUnion(
                r, new Range(xlat(c), xlat(c) + 1));
        return r;
    }
}
```

```

}
}

```

In this code, the inner loop "for (c = lb + 1; c <= ub; c++)" is arranged by the "c" variable. The outer loop also uses the "c" variable. But there is no error here. After the inner loop is executed, the "return r;" operator will perform exit from the function. Unfortunately, the analyzer cannot make this situation out. You may use a different variable for the inner loop or suppress the warning with the "//V535" comment.

V536. Be advised that the utilized constant value is represented by an octal form.

Using constants in the octal number system is not an error in itself. This system is convenient when handling bits and is used in code that interacts with a network or external devices. However, an average programmer uses this number system rarely and therefore may make a mistake by writing 0 before a number forgetting that it makes this value an octal number.

The PVS-Studio analyzer warns about octal constants if there are no other octal constants nearby. Such "single" octal constants are usually errors.

Let's study a sample taken from a real application. It is rather large but it illustrates the sense of the issue very well.

```

inline
void elxLuminocity(const PixelRGBf& iPixel,
                  LuminanceCell< PixelRGBf >& oCell)
{
    oCell._luminance = 0.2220f*iPixel._red +
                      0.7067f*iPixel._blue +
                      0.0713f*iPixel._green;
    oCell._pixel = iPixel;
}

inline
void elxLuminocity(const PixelRGBi& iPixel,
                  LuminanceCell< PixelRGBi >& oCell)
{
    oCell._luminance = 2220*iPixel._red +
                      7067*iPixel._blue +
                      0713*iPixel._green;
    oCell._pixel = iPixel;
}

```

It is hard to find the error while reviewing this code, but it does have an error. The first function elxLuminocity is correct and handles values of the 'float' type. There are the following constants in the code: 0.2220f, 0.7067f, 0.0713f. The second function is similar to the first but it handles integer values. All the integer values are multiplied by 10000. Here are they: 2220, 7067, 0713. The error is that the last constant "0713" is defined in the octal number system and its value is 459, not 713. This is the correct code:

```

oCell._luminance = 2220*iPixel._red +
                  7067*iPixel._blue +
                  713*iPixel._green;

```

As it was said above, the warning of octal constants is generated only if there are no other octal constants nearby. That is why the analyzer considers the following sample safe and does not produce any warnings for it:

```

static unsigned short bytebit[8] = {
    01, 02, 04, 010, 020, 040, 0100, 0200 };

```

V537. Consider reviewing the correctness of 'X' item's usage.

The analyzer detected a potential misprint in code. This rule tries to diagnose an error of the following type using the heuristic method:

```

int x = static_cast<int>(GetX()) * n;
int y = static_cast<int>(GetX()) * n;

```

In the second line, the GetX() function is used instead of GetY(). This is the correct code:

```

int x = static_cast<int>(GetX()) * n;
int y = static_cast<int>(GetY()) * n;

```

To detect this suspicious fragment, the analyzer followed this logic: we have a line containing a name that includes the "X" fragment. Beside it, there is a line that has an antipode name with "Y". But this second line has "X" as well. Since this condition and some other conditions hold, the construct must be reviewed by the programmer. This code would not be considered dangerous if, for instance, there were no variables "x" and "y" to the left. This is a code sample the analyzer ignores:

```

array[0] = GetX() / 2;
array[1] = GetX() / 2;

```

Unfortunately, this rule often produces false alarms since the analyzer does not know how the program is organized and what the code's purpose is. This is a sample of a false alarm:

```

halfWidth -= borderWidth + 2;
halfHeight -= borderWidth + 2;

```

The analyzer supposed that the second line must be presented by a different expression, for instance, "halfHeight -= borderHeight + 2". But actually there is no error here. The border's size is equal in both vertical and horizontal positions. There is just no borderHeight constant. However, such high-level abstractions are not clear to the analyzer. To suppress this warning, you may type the "//V537" comment into the code.

V538. The line contains control character 0x0B (vertical tabulation).

There are ASCII control characters in the program text. The following character refers to them:

0x0B - LINE TABULATION (vertical tabulation) - Moves the typing point to the next vertical tabulation position. In terminals, this character is usually equivalent to line feed.

Such characters are allowed to be present in program text and such text is successfully compiled in Visual C++. However, these characters must have appeared in the program text by accident and you'd better get rid of them. There are two reasons for that:

1) If such a control character stands in the first lines of a file, the Visual Studio environment cannot understand the file's format and opens it with the Notepad application instead of its own embedded editor.

2) Some external tools working with program texts may incorrectly process files containing the above mentioned control characters.

0x0B characters are invisible in the Visual Studio 2010 editor. To find and delete them in a line, you may open the file in the Notepad application or any other editor that can display such control characters.

V539. Consider inspecting iterators which are being passed as arguments to function 'Foo'.

The analyzer detected code handling containers which is likely to have an error. You should examine this code fragment.

Let's study several samples demonstrating cases when this warning is generated:

Sample 1.

```
void X(std::vector<int> &X, std::vector<int> &Y)
{
    std::for_each (X.begin(), X.end(), SetValue);
    std::for_each (Y.begin(), X.end(), SetValue);
}
```

Two arrays are filled with some values in the function. Due to the misprint, the "std::for_each" function, being called for the second time, receives iterators from different containers, which causes an error during program execution. This is the correct code:

```
std::for_each (X.begin(), X.end(), SetValue);
std::for_each (Y.begin(), Y.end(), SetValue);
```

Sample 2.

```
std::includes(a.begin(), a.end(), a.begin(), a.end());
```

This code is strange. The programmer most probably intended to process two different chains instead of one. This is the correct code:

```
std::includes(a.begin(), a.end(), b.begin(), b.end());
```

V540. Member 'x' should point to string terminated by two 0 characters.

In Windows API, there are structures where string-pointers must end with a double zero. For example, such is the lpstrFilter member in the OPENFILENAME structure.

Here is the description of lpstrFilter in MSDN:

LPCTSTR

A buffer containing pairs of null-terminated filter strings. The last string in the buffer must be terminated by two NULL characters.

It follows from this description that we must add one more zero at the end of the string. For example: lpstrFilter = "All Files\0*.*\0";

However, many programmers forget about this additional zero. This is a sample of incorrect code we found in one application:

```
lofn.lpstrFilter = L"Equalizer Preset (*.feq)\0*.feq";
```

This code will cause generating rubbish in the filter field in the file dialogue. This is the correct code:

```
lofn.lpstrFilter = L"Equalizer Preset (*.feq)\0*.feq\0";
```

We added 0 at the end of the string manually while the compiler will add one more zero. Some programmers write this way to make it clearer:

```
lofn.lpstrFilter = L"Equalizer Preset (*.feq)\0*.feq\0\0";
```

But here we will get three zeroes instead of two. It is unnecessary yet well visible to the programmer.

There are also some other structures besides OPENFILENAME where you might make such mistakes. For instance, the strings lpstrGroupNames and lpstrCardNames in structures OPENCARD_SEARCH_CRITERIA, OPENCARDNAME must end with a double zero too.

V541. It is dangerous to print the string into itself.

The analyzer detected a potential error: a string gets printed inside itself. This may lead to unexpected results. Look at this sample:

```
char s[100] = "test";
sprintf(s, "N = %d, S = %s", 123, s);
```

In this code, the 's' buffer is used simultaneously as a buffer for a new string and as one of the elements making up the text. The programmer intends to get this string:

N = 123, S = test

But actually this code will cause creating the following string:

N = 123, S = N = 123, S =

In other cases, such code may cause even a program crash. To fix the code, we should use a new buffer to save the result. This is the correct code:

```
char s1[100] = "test";
char s2[100];
sprintf(s2, "N = %d, S = %s", 123, s1);
```

V542. Consider inspecting an odd type cast: 'Type1' to 'Type2'.

The analyzer found a very suspicious explicit type conversion. This type conversion may signal an error. You should review the corresponding code fragment.

For example:

```
typedef unsigned char Byte;

void Process(wchar_t ch);
void Process(wchar_t *str);

void Foo(Byte *buf, size_t nCount)
{
```

```

for (size_t i = 0; i < nCount; ++i)
{
    Process((wchar_t *)buf[i]);
}

```

There is the Process function that can handle both separate characters and strings. There is also the 'Foo' function which receives a buffer-pointer at the input. This buffer is handled as an array of characters of the wchar_t type. But the code contains an error, so the analyzer warns you that the 'char' type is explicitly cast to the 'wchar_t*' type. The reason is that the "(wchar_t *)buf[i]" expression is equivalent to "(wchar_t *)(&buf[i])". A value of the 'char' type is first fetched out of the array and then cast to a pointer. This is the correct code:

```
Process((wchar_t *)buf[i]);
```

However, strange type conversions are not always errors. Consider a sample of safe code taken from a real application:

```

wchar_t *destStr = new wchar_t[1024];
...
for (int j = 0 ; j < nbChar ; j++)
{
    if (Case == UPPERCASE)
        destStr[j] =
            (wchar_t)::CharUpperW((LPWSTR)destStr[j]);
    ...
}

```

Here you may see an explicit conversion of the 'wchar_t' type to 'LPWSTR' and vice versa. The point is that Windows API and the CharUpperW function can handle an input value both as a pointer and a character. This is the function's prototype:

```
LPTSTR WINAPI CharUpperW(__inout LPWSTR lpsz);
```

If the high-order part of the pointer is 0, the input value is considered a character. Otherwise, the function processes the string.

The analyzer knows about the CharUpperW function's behavior and considers this code safe. But it may produce a false alarm in some other similar situation.

V543. It is odd that value 'X' is assigned to the variable 'Y' of HRESULT type.

The analyzer detected a potential error related to handling a variable of the HRESULT type.

HRESULT is a 32-bit value divided into three different fields: severity code, device code and error code. Such special constants as S_OK, E_FAIL, E_ABORT, etc. serve to handle HRESULT-values while the SUCCEEDED and FAILED macros are used to check HRESULT-values.

The V543 warning is generated if the analyzer detects an attempt to write value -1, true or false into a variable of the HRESULT type. Consider this sample:

```

HRESULT h;
...
if (bExceptionCaught)
{
    ShowPluginErrorMessage(pi, errorText);
    h = -1;
}

```

Writing of value "-1" is incorrect. If you want to report about some unspecified error, you should use value 0x80004005L (Unspecified failure). This constant and the like are described in "WinError.h". This is the correct code:

```

if (bExceptionCaught)
{
    ShowPluginErrorMessage(pi, errorText);
    h = E_FAIL;
}

```

References:

1. MSDN. Common HRESULT Values. <http://www.viva64.com/go.php?url=421>
2. Wikipedia. HRESULT. <http://www.viva64.com/go.php?url=422>

V544. It is odd that the value 'X' of HRESULT type is compared with 'Y'.

The analyzer detected a potential error related to handling a variable of the HRESULT type.

HRESULT is a 32-bit value divided into three different fields: severity code, device code and error code. Such special constants as S_OK, E_FAIL, E_ABORT, etc. serve to handle HRESULT-values while the SUCCEEDED and FAILED macros are used to check HRESULT-values.

The V544 warning is generated if the analyzer detects an attempt to compare a variable of the HRESULT type to -1, true or false. Consider this sample:

```

HRESULT hr;
...
if (hr == -1)
{
}

```

Comparison of the variable to "-1" is incorrect. Error codes may differ. For instance, these may be 0x80000002L (Ran out of memory), 0x80004005L (unspecified failure), 0x80070005L (General access denied error) and so on. To check the HRESULT -value in this case, we must use the FAILED macro defined in "WinError.h". This is the correct code:

```

if (FAILED(hr))
{
}

```

References:

1. MSDN. Common HRESULT Values. <http://www.viva64.com/go.php?url=421>
2. Wikipedia. HRESULT. <http://www.viva64.com/go.php?url=422>

V545. Such conditional expression of 'if' operator is incorrect for the HRESULT type value 'Foo'. The SUCCEEDED or FAILED macro should be used instead.

The analyzer detected a potential error related to handling a variable of the HRESULT type.

HRESULT is a 32-bit value divided into three different fields: severity code, device code and error code. Such special constants as S_OK, E_FAIL, E_ABORT, etc. serve to handle HRESULT-values while the SUCCEEDED and FAILED macros are used to check HRESULT-values.

The V545 warning is generated if a variable of the HRESULT type is used in the 'if' operator as a bool-variable. Consider this sample:

```
HRESULT hr;
...
if (hr)
{
}
```

'HRESULT' and 'bool' are two types absolutely different in meaning. This sample of comparison is incorrect. The HRESULT type can have many states including 0L (S_OK), 0x80000002L (Ran out of memory), 0x80004005L (unspecified failure) and so on. Note that the code of the state S_OK is 0.

To check the HRESULT-value, we must use macro SUCCEEDED or FAILED defined in "WinError.h". These are correct versions of code:

```
if (FAILED(hr))
{
}
if (SUCCEEDED(hr))
{
}
```

References:

1. MSDN. Common HRESULT Values. <http://www.viva64.com/go.php?url=421>
2. Wikipedia. HRESULT. <http://www.viva64.com/go.php?url=422>

V546. Member of a class is initialized by itself: 'Foo(Foo)'.

The analyzer detected a misprint in the fragment when a class member is being initialized by itself. Consider an example of a constructor:

```
C95(int field) : Field(Field)
{
    int Field;
    ...
}
```

The names of the argument and the class member here differ only in one letter. Because of that, the programmer misprinted here causing the Field member to remain uninitialized. This is the correct code:

```
C95(int field) : Field(field)
{
    int Field;
    ...
}
```

V547. Expression is always true/false.

The analyzer detected a potential error: a condition is always true or false. Such conditions do not always signal an error but still you must review such code fragments.

Consider a code sample:

```
LRESULT CALLBACK GridProc(HWND hWnd,
    UINT message, WPARAM wParam, LPARAM lParam)
{
    ...
    if (wParam<0)
    {
        BGHS[SelfIndex].rows = 0;
    }
    else
    {
        BGHS[SelfIndex].rows = MAX_ROWS;
    }
    ...
}
```

The "BGHS[SelfIndex].rows = 0;" branch here will never be executed because the wParam variable has an unsigned type WPARAM which is defined as "typedef UINT_PTR WPARAM".

Either this code contains a logical error or we may reduce it to one line: "BGHS[SelfIndex].rows = MAX_ROWS;".

Now let's examine a code sample which is correct yet potentially dangerous and contains a meaningless comparison:

```
unsigned int a = _ttoi(LPCTSTR(str1));
if((0 > a) || (a > 255))
{
    return(FALSE);
}
```

The programmer wanted to implement the following algorithm.

- 1) Convert a string into a number.
- 2) If the number lies outside the range [0..255], return FALSE.

The error here is in using the 'unsigned' type. If the _ttoi function returns a negative value, it will turn into a large positive value. For instance, value "-3" will become 4294967293. The comparison '0 > a' will always return true. The program works correctly only because the range of values [0..255] is checked by the 'a > 255' condition.

The analyzer will generate the following warning for this code fragment: "V547 Expression '0 > a' is always false. Unsigned type value is never < 0."

We should correct this fragment this way:

```
int a = _ttoi(LPCTSTR(str1));
if((0 > a) || (a > 255))
{
    return(FALSE);
}
```



```
}
```

Let's consider one special case. The analyzer generates the warning:

V547 Expression 's == "Abcd"' is always false. To compare strings you should use strcmp() function.

for this code:

```
const char *s = "Abcd";
void Test()
{
    if (s == "Abcd")
        cout << "TRUE" << endl;
    else
        cout << "FALSE" << endl;
}
```

But it is not quite true. This code still can print "TRUE" when the 's' variable and Test() function are defined in one module. The compiler does not produce a lot of identical constant strings but uses one string. As a result, the code sometimes seems quite operable. However, you must understand that this code is very bad and you should use special functions for comparison.

The analyzer warns you far not of all the conditions which are always false or true. It diagnoses only those cases when an error is highly probable. Let's consider some samples that the analyzer considers absolutely correct:

```
// 1) Eternal loop
while (true)
{
    ...
}

// 2) Macro expanded in the Release version
// MY_DEBUG_LOG("X=", x);
0 && ("X=", x);

// 3) assert(false);
if (error) {
    assert(false);
    return -1;
}
```

V548. Consider reviewing type casting. TYPE X[][] in not equivalent to TYPE **X.

The analyzer detected a potential error related to an explicit type conversion. An array defined as "type Array[3][4]" is cast to type "type ***". This type conversion is most likely to be meaningless.

Types "type[a][b]" and "type ***" are different data structures. Type[a][b] is a single memory area that you can handle as a two-dimensional array. Type ** is an array of pointers referring to some memory areas.

Here is an example:

```
void Foo(char **names, size_t count)
{
    for(size_t i=0; i<count; i++)
        printf("%s\n", names[i]);
}

void Foo2()
{
    char names[32][32];
    ...
    F103_((char **)names, 32); //Crash
}
```

This is the correct code:

```
void Foo2()
{
    char names[32][32];
    ...
    char *names_p[32];
    for(size_t i=0; i<32; i++)
        names_p[i] = names[i];
    F103_(names_p, 32); //OK
}
```

V549. The 'first' argument of 'Foo' function matches it's the 'second' argument.

The analyzer detected a potential error in the program: coincidence of two actual arguments of a function. Passing the same value as two arguments is a normal thing for many functions. But if you deal with such functions as memmove, memcpy, strstr and strcmp, you must check the code.

Here is a sample from a real application:

```
#define str_cmp(s1, s2) wscmp(s1, s2)
...
v = abs(str_cmp(a->tdata, a->tdata));
```

The misprint here causes the wscmp function to perform comparison of a string from itself. This is the correct code:

```
v = abs(str_cmp(a->tdata, b->tdata));
```

The analyzer generates the warning if the following functions are being handled: memcpy, memmove, memcmp, _memicmp, strstr, strspn, strtok, strcmp, strncmp, wscmp, _stricmp, wcsncmp, etc. If you found a similar error that PVS-Studio fails to diagnose, please tell us the name of the function that must not take same values as the first and second arguments.

V550. An odd precise comparison. It's probably better to use a comparison with defined precision: fabs(A - B) < Epsilon or fabs(A - B) >

Epsilon.

The analyzer detected a potential error: the == or != operator is used to compare floating point numbers. Precise comparison might often cause an error.

Consider this sample:

```
double a = 0.5;
if (a == 0.5) //OK
    x++;

double b = sin(M_PI / 6.0);
if (b == 0.5) //ERROR
    x++;
```

The first comparison 'a == 0.5' is true. The second comparison 'b == 0.5' may be both true and false. The result of the 'b == 0.5' expression depends upon the processor, compiler's version and settings being used. For instance, the 'b' variable's value was 0.49999999999999994 when we used the Visual C++ 2010 compiler. A more correct version of this code looks this way:

```
double b = sin(M_PI / 6.0);
if (fabs(b - 0.5) < DBL_EPSILON)
    x++;
```

In this case, the comparison with error presented by DBL_EPSILON is true because the result of the sin() function lies within the range [-1, 1]. But if we handle values larger than several units, errors like FLT_EPSILON and DBL_EPSILON will be too small. And vice versa, if we handle values like 0.00001, these errors will be too big. Each time you must choose errors adequate to the range of possible values.

Question: how do I compare two double-variables then?

```
double a = ...;
double b = ...;
if (a == b) // how?
{
}
```

There is no single right answer. In most cases, you may compare two variables of the double type by writing the following code:

```
if (fabs(a-b) <= DBL_EPSILON * fmax(fabs(a), fabs(b)))
{
}
```

But be careful with this formula - it works only for numbers with the same sign. Besides, if you have a row with many calculations, there is an error constantly accumulating, which might cause the DBL_EPSILON constant to appear a too small value.

Well, can I perform precise comparison of floating point values?

Sometimes, yes. But rather rarely. You may perform such comparison if the values you are comparing are one and the same value in its sense.

Here is a sample where you may use precise comparison:

```
// -1 - value is not initialized.
float val = -1.0f;
if (Pool())
    val = 123.0f;
if (val == -1.0f) //OK
{
}
```

In this case, the comparison with value "-1" is permissible because it is this very value which we used to initialize the variable before.

We cannot cover the topic of comparing float/double types within the scope of documentation, so please refer to additional sources given at the end of this article.

The analyzer can only point to potentially dangerous code fragments where comparison may result unexpectedly. But it is only the programmer who may understand whether these code fragments really contain errors. We cannot also give precise recommendations in the documentation since tasks where floating point types are used are too diverse.

The diagnostic message isn't generated if two identical expressions of 'float' or 'double' types are being compared. Such a comparison allows to identify the value as NaN. The example of code implementing the verification of this kind:

```
bool isnan(double X) { return X != X; }
```

References:

1. Bruce Dawson. Comparing floating point numbers. <http://www.viva64.com/go.php?url=423>
2. Viva64 Blog. 64-bit programs and floating-point calculations. <http://www.viva64.com/en/b/0074/>
3. Wokipedia. Floating point. <http://www.viva64.com/go.php?url=425>
4. CodeGuru Forums. C++ General: How is floating point represented? <http://www.viva64.com/go.php?url=426>

V551. The code under this 'case' label is unreachable.

The analyzer detected a potential error: one of the switch() operator's branches never gets control. The reason is that the switch() operator's argument cannot accept the value defined in the case operator. Consider this sample:

```
char ch = strText[i];
switch (ch)
{
case '<':
    strHTML += "&lt;";
    bLastCharSpace = FALSE;
    nNonbreakChars++;
    break;
case '>':
    strHTML += "&gt;";
    bLastCharSpace = FALSE;
    nNonbreakChars++;
    break;
case 0xB7:
case 0xBB:
    strHTML += ch;
    strHTML += "<wbr>";
    bLastCharSpace = FALSE;
    nNonbreakChars = 0;
    break;
...
}
```

```
}
```

The branch following "case 0xB7:" and "case 0xBB:" in this code will never get control. The 'ch' variable has the 'char' type and therefore the range of its values is [-128..127]. The comparisons "ch == 0xB7" and "ch==0xBB" will always be false. To make the code correct, we must cast the 'ch' variable to the 'unsigned char' type:

```
unsigned char ch = strText[i];
switch (ch)
{
...
case 0xB7:
case 0xBB:
    strHTML += ch;
    strHTML += "<wbr>";
    bLastCharSpace = FALSE;
    nNonbreakChars = 0;
    break;
...
}
```

V552. A bool type variable is being incremented. Perhaps another variable should be incremented instead.

The analyzer detected a potentially dangerous construct in code where a variable of the bool type is being incremented:

```
bool bValue = false;
...
bValue++;
```

First, the C++ language's standard reads:

The use of an operand of type bool with the postfix ++ operator is deprecated.

It means that we should not use such a construct.

Second, it is better to assign the 'true' value explicitly to this variable. This code is clearer:

```
bValue = true;
```

Third, it might be that there is a misprint in the code and the programmer actually intended to increment a different variable. For example:

```
bool bValue = false;
int iValue = 1;
...
if (bValue)
    bValue++;
```

A wrong variable was used by accident here while it was meant to be this code:

```
bool bValue = false;
int iValue = 1;
...
if (bValue)
    iValue++;
```

V553. The length of function's body or class's declaration is more than 2000 lines long. You should consider refactoring the code.

The analyzer detected a class definition or function body that occupies more than 2000 lines. This class or function does not necessarily contain errors yet the probability is very high. The larger a function is, the more probable it is to make an error and the more difficult it is to debug. The larger a class is, the more difficult it is to examine its interfaces.

This message is a good opportunity to find time for code refactoring at last. Yes, you always have to do something urgent but the larger you functions and classes are, the more time you will spend on supporting the old code and eliminating errors in it instead of writing a new functionality.

References:

1. Steve McConnell, "Code Complete, 2nd Edition" Microsoft Press, Paperback, 2nd edition, Published June 2004, 914 pages, ISBN: 0-7356-1967-0. (**Part 7.4. How Long Can a Routine Be?**).

V554. Incorrect use of smart pointer.

Analyzer located an issue then the usage of smart pointer could lead to undefined behavior, in particular to the heap damage, abnormal program termination or incomplete objects destruction. The error here is that different methods will be used to allocate and free memory.

Consider a sample:

```
void Foo()
{
    struct A
    {
        A() { cout << "A()" << endl; }
        ~A() { cout << "~A()" << endl; }
    };

    std::unique_ptr<A> p(new A[3]);
}
```

By default, the unique_ptr class uses the 'delete' operator to release memory. That is why only one object of the 'A' class will be destroyed and the following text will be displayed:

```
A()
A()
A()
~A()
```

To fix this error, we must specify that the class must use the 'delete []' operator. Here is the correct code:

```
std::unique_ptr<A[]> p(new A[3]);
```

Now the same number of constructors and destructors will be called and we will see this text:

```
A()
A()
A()
~A()
~A()
~A()
```

Consider another sample:

```
std::unique_ptr<int []> p((int *)malloc(sizeof(int) * 5));
```

The function 'malloc()' is used to allocate memory while the 'delete []' operator is used to release it. It is incorrect and we must specify that the 'free()' function must be used to release memory. This is the correct code:

```
int *d=(int *)std::malloc(sizeof(int) * 5);
unique_ptr<int, void (*)(void*)> p(d, std::free);
```

Additional materials on this topic:

1. Discussion at StackOverflow. "How could pairing new[] with delete possibly lead to memory leak only?". <http://www.viva64.com/go.php?url=662>

V555. The expression of the 'A - B > 0' kind will work as 'A != B'.

The analyzer detected a potential error in an expression of "A - B > 0" type. It is highly probable that the condition is wrong if the "A - B" subexpression has the unsigned type.

The "A - B > 0" condition holds in all the cases when 'A' is not equal to 'B'. It means that we may write the "A != B" expression instead of "A - B > 0". However, the programmer must have intended to implement quite a different thing.

Consider this sample:

```
unsigned int *B;
...
if (B[i]-70 > 0)
```

The programmer wanted to check whether the i-item of the B array is above 70. He could write it this way: "B[i] > 70". But he, proceeding from some reasons, wrote it this way: "B[i]-70 > 0" and made a mistake. He forgot that items of the 'B' array have the 'unsigned' type. It means that the "B[i]-70" expression has the 'unsigned' type too. So it turns out that the condition is always true except for the case when the 'B[i]' item equals to 70.

Let's clarify this case.

If 'B[i]' is above 70, then "B[i]-70" is above 0.

If 'B[i]' is below 70, then we will get an overflow of the unsigned type and a very large value as a result. Let B[i] == 50. Then "B[i]-70" = 50u - 70u = 0xFFFFFECu = 4294967276. Surely, 4294967276 > 0.

A demonstration sample:

```
unsigned A;
A = 10; cout << "A=10 " << (A-70 > 0) << endl;
A = 70; cout << "A=70 " << (A-70 > 0) << endl;
A = 90; cout << "A=90 " << (A-70 > 0) << endl;
// Will be printed
A=10 1
A=70 0
A=90 1
```

The first way to correct the code:

```
unsigned int *B;
...
if (B[i] > 70)
```

The second way to correct the code:

```
int *B;
...
if (B[i]-70 > 0)
```

Note that an expression of the "A - B > 0" type far not always signals an error. Consider a sample where the analyzer generates a false alarm:

```
// Functions GetLength() and GetPosition() return
// value of size_t type.
while ((inStream.GetLength() - inStream.GetPosition()) > 0)
{ ... }
```

GetLength() is always above or equal to GetPosition() here, so the code is correct. To suppress the false alarm, we may add the comment //-V555 or rewrite the code in the following way:

```
while (inStream.GetLength() != inStream.GetPosition())
{ ... }
```

Here is another case when no error occurs.

```
__int64 A;
__uint32 B;
...
if (A - B > 0)
```

The "A - B" subexpression here has the signed type __int64 and no error occurs. The PVS-Studio analyzer does not generate warnings in such cases.

V556. The values of different enum types are compared.

The analyzer detected a potential error: code contains comparison of enum values which have different types.

Consider a sample:

```
enum ErrorTypeA { E_OK, E_FAIL };
enum ErrorTypeB { E_ERROR, E_SUCCESS };
void Foo(ErrorTypeB status) {
```

```

    if (status == E_OK)
    { ... }
}

```

The programmer used a wrong name in the comparison by accident, so the program's logic is disrupted. This is the correct code:

```

void Foo(ErrorTypeB status) {
    if (status == E_SUCCESS)
    { ... }
}

```

Comparison of values of different enum types is not necessarily an error, but you must review such code.

V557. Array overrun is possible.

The analyzer detected a potential memory access outside an array. The most common case is an error occurring when writing the '\0' character after the last array's item. Let's examine a sample of this error:

```

struct IT_SAMPLE
{
    unsigned char filename[14];
    ...
};

static int it_riff_dsmf_process_sample(
    IT_SAMPLE * sample, const unsigned char * data)
{
    memcpy( sample->filename, data, 13 );
    sample->filename[ 14 ] = 0;
    ...
}

```

The last array's item has index 13, not 14. That is why the correct code is this one:

```
sample->filename[13] = 0;
```

Of course, you'd better use an expression involving the sizeof() operator instead of constant index' value in such cases. However, remember that you may make a mistake in this case too. For example:

```

typedef wchar_t letter;
letter    name[30];
...
name[sizeof(name) - 1] = L'\0';

```

At first sight, the "sizeof(name) - 1" expression is right. But the programmer forgot that he handled the 'wchar_t' type and not 'char'. As a result, the '\0' character is written far outside the array's boundaries. This is the correct code:

```
name[sizeof(name) / sizeof(*name) - 1] = L'\0';
```

To simplify writing of such constructs, you may use this special macro:

```

#define str_len(arg) ((sizeof(arg) / sizeof(arg[0])) - 1)
name[str_len(name)] = L'\0';

```

The analyzer detects some errors when the index is represented by a variable whose value might run out of the array's boundaries. For example:

```

int buff[25];
for (int i=0; i <= 25; i++)
    buff[i] = 10;

```

This is the correct code:

```

int buff[25];
for (int i=0; i < 25; i++)
    buff[i] = 10;

```

Note that the analyzer might make mistakes when handling such value ranges and generate false alarms.

V558. Function returns the pointer to temporary local object.

The analyzer detected an issue when a function returns a pointer to a local object. This object will be destroyed when leaving the function, so you will not be able to use the pointer to it anymore. In a most common case, this diagnostic message is generated against the following code:

```

float *F()
{
    float f = 1.0;
    return &f;
}

```

Of course, the error would hardly be present in such a form in real code. Let's consider a more real example.

```

int *Foo()
{
    int A[10];
    // ...
    if (err)
        return 0;

    int *B = new int[10];
    memcpy(B, A, sizeof(A));

    return A;
}

```

Here, we handled the temporary array A. On some condition, we must return the pointer to the new array B. But the misprint causes the A array to be returned, which will cause unexpected behavior of the program or crash. This is the correct code:

```

int *Foo()
{
    ...
    int *B = new int[10];
    memcpy(B, A, sizeof(A));
    return B;
}

```

V559. Suspicious assignment inside the condition expression of 'if/while/for' operator.

The analyzer detected an issue when an 'if' or 'while' operator's condition contains the assignment operator '=' while the variable is assigned a constant value. Such a construct usually signals an error. It is highly probable that the programmer intended to use the '==' operator instead of '='.

Consider this sample:

```
const int MAX_X = 100;
int x;
...
if (x = MAX_X)
{ ... }
```

There is a misprint in the code. Instead of comparing the 'x' variable with the MAX_X constant, the value of the 'x' variable will be changed. This is the correct code:

```
if (x == MAX_X)
{ ... }
```

If a variable is assigned a non-constant value, the V559 warning will not be generated. Assignment inside a condition is a frequent method to abridge the code. Here is a code sample the analyzer considers safe:

```
while (x = get())
{
    ...
}
```

V560. A part of conditional expression is always true/false.

The analyzer detected a potential error inside a logical condition. A part of a logical condition is always true and therefore is considered dangerous.

Consider this sample:

```
#define REO_INPLACEACTIVE (0x02000000L)
...
if (reObj.dwFlags && REO_INPLACEACTIVE)
    m_pRichEditOle->InPlaceDeactivate();
```

The programmer wanted to check some particular bit in the dwFlags variable. But he made a misprint by writing the '&&' operator instead of '&' operator. This is the correct code:

```
if (reObj.dwFlags & REO_INPLACEACTIVE)
    m_pRichEditOle->InPlaceDeactivate();
```

Let's examine another sample:

```
if (a = 10 || a == 20)
```

The programmer wrote the assignment operator '=' instead of comparison operator '==' by accident. From the viewpoint of the C++ language, this expression is identical to an expression like "if (a = (10 || a == 20))".

The analyzer considers the "10 || a == 20" expression dangerous because its left part is a constant. This is the correct code:

```
if (a == 10 || a == 20)
```

Some C++ constructs are considered safe even if a part of an expression inside them is a constant. Here are some samples when the analyzer considers the code safe:

- a subexpression contains operators sizeof(): if (a == b && sizeof(T) < sizeof(__int64)) {};
- an expression is situated inside a macro: assert(false);
- two numerical constants are being compared: if (MY_DEFINE_BITS_COUNT == 4) {};
- etc.

V561. It's probably better to assign value to 'foo' variable than to declare it anew.

The analyzer detected a potential error: there is a variable in code which is defined and initialized but not being used further. Besides, there is a variable in the exterior scope which also has the same name and type. It is highly probable that the programmer intended to use an already existing variable instead of defining a new one.

Let's examine this sample:

```
BOOL ret = TRUE;
if (m_hbitmap)
    BOOL ret = picture.SaveToFile(fpPtr);
```

The programmer defined a new variable 'ret' by accident, which causes the previous variable to always have the TRUE value regardless if the picture is saved into a file successfully or not. This is the correct code:

```
BOOL ret = TRUE;
if (m_hbitmap)
    ret = picture.SaveToFile(fpPtr);
```

V562. It's odd to compare a bool type value with a value of N.

The analyzer detected an issue when a value of the bool type is compared to a number not equal to 0 or 1. Most likely, there is an error.

Consider this sample:

```
if (0 < A < 5)
```

The programmer not familiar with the C++ language well wanted to use this code to check whether the value lies within the range between 0 and 5. Actually, the calculations will be performed in the following sequence: ((0 < A) < 5). The result of the "0 < A" expression has the bool type and therefore is always below 5.

This is the correct code for the check:


```
if (0 < A && A < 5)
```

The previous example resembles a mistake usually made by students. But even skilled developers are not secure from such errors.

Let's consider another sample:

```
if (! (fp = fopen(filename, "wb")) == -1) {
    perror("opening image file failed");
    exit(1);
}
```

Here we have 2 errors of different types at once. First, the "fopen" function returns the pointer and compares the returned value to NULL. The programmer confused the "fopen" function with "open" function, the latter being that very function that returns "-1" if there is an error. Second, the negation operation "!" is executed first and only then the value is compared to "-1". There is no sense in comparing a value of the bool type to "-1" and that is why the PVS-Studio analyzer warned us about the code.

This is the correct code:

```
if ( (fp = fopen(filename, "wb")) == NULL) {
    perror("opening image file failed");
    exit(1);
}
```

V563. It is possible that this 'else' branch must apply to the previous 'if' statement.

The analyzer detected a potential error in logical conditions: code's logic does not coincide with the code editing.

Consider this sample:

```
if (X)
    if (Y) Foo();
else
    z = 1;
```

The code editing disorients you so it seems that the "z = 1" assignment takes place if X == false. But the 'else' branch refers to the nearest operator 'if'. In other words, this code is actually analogous to the following code:

```
if (X)
{
    if (Y)
        Foo();
    else
        z = 1;
}
```

So, the code does not work the way it seems at first sight.

If you get the V563 warning, it may mean one of the two following things:

1) Your code is badly edited and there is no error actually. In this case you need to edit the code so that it becomes clearer and the V563 warning is not generated. Here is a sample of correct editing:

```
if (X)
    if (Y)
        Foo();
else
    z = 1;
```

2) A logical error has been found. Then you may correct the code, for instance, this way:

```
if (X) {
    if (Y)
        Foo();
} else {
    z = 1;
}
```

V564. The '&' or '|' operator is applied to bool type value. You've probably forgotten to include parentheses or intended to use the '&&' or '||' operator.

The analyzer detected a potential error: operators '&' and '|' handle bool-type values. Such expressions are not necessarily errors but they usually signal misprints or condition errors.

Consider this sample:

```
int a, b;
#define FLAG 0x40
...
if (a & FLAG == b)
{
}
```

This example is a classic one. A programmer may be easily mistaken in operations' priorities. It seems that computing runs in this sequence: "(a & FLAG) == b". But actually it is "a & (FLAG == b)". Most likely, it is an error.

The analyzer will generate a warning here because it is odd to use the '&' operator for variables of int and bool types.

If it turns out that the code does contain an error, you may fix it the following way:

```
if ((a & FLAG) == b)
```

Of course, the code might appear correct and work as it was intended. But still you'd better rewrite it to make it clearer. Use the '&&' operator or additional brackets:

```
if (a && FLAG == b)
if (a & (FLAG == b))
```

The V564 warning will not be generated after these corrections are done while the code will get easier to read.

Consider another sample:

```
#define SVF_CASTAI 0x00000010
if ( !ent->r.svFlags & SVF_CASTAI ) {
    ...
}
```

Here we have an obvious error. It is the "!ent->r.svFlags" subexpression that will be calculated at first and we will get either true or false. But it does not matter: whether we execute "true & 0x00000010" operation or "false & 0x00000010" operation, the result will be the same. The condition in this sample is always false.

This is the correct code:

```
if ( ! (ent->r.svFlags & SVF_CASTAI) )
```

Note. The analyzer will not generate the warning if there are bool-type values to the left and to the right of the '&' or '|' operator. Although such code does not look too smart, still it is correct. Here is a code sample the analyzer considers safe:

```
bool X, Y;
...
if (X | Y)
{ ... }
```

V565. An empty exception handler. Silent suppression of exceptions can hide the presence of bugs in source code during testing.

An exception handler was found that does not do anything. Consider this code:

```
try {
    ...
}
catch (MyExcept &)
{
}
```

Of course, this code is not necessarily incorrect. But it is very odd to suppress an exception by doing nothing. Such exception handling might conceal defects in the program and complicate the testing process.

You must react to exceptions somehow. For instance, you may add "assert(false)" at least:

```
try {
    ...
}
catch (MyExcept &)
{
    assert(false);
}
```

Programmers sometimes use such constructs to return control from a number of nested loops or recursive functions. But it is bad practice because exceptions are very resource-intensive operations. They must be used according to their intended purpose, i.e. for possible contingencies that must be handled on a higher level.

The only thing where you may simply suppress exceptions is destructors. A destructor must not throw exceptions. But it is often not quite clear what to do with exceptions in destructors and the exception handler might well remain empty. The analyzer does not warn you about empty handlers inside destructors:

```
CClass::~CClass()
{
    try {
        DangerousFreeResource();
    }
    catch (...) {
    }
}
```

V566. The integer constant is converted to pointer. Possibly an error or a bad coding style.

The analyzer detected an explicit conversion of a numerical value to the pointer type. This warning is usually generated for code fragments where numbers are used for flagging objects' states. Such methods are not necessarily errors but usually signal a bad code design. Consider this sample:

```
const DWORD SHELL_VERSION = 0x4110400;
...
char *ptr = (char*) SHELL_VERSION;
...
if (ptr == (char*) SHELL_VERSION)
```

The constant value which marks some special state is saved into the pointer. This code might work well for a long time, but if an object is created by the address 0x4110400, we will not determine if this is a magic flag or just an object. If you want to use a special flag, you'd better write it so:

```
const DWORD SHELL_VERSION = 0x4110400;
...
char *ptr = (char*) (&SHELL_VERSION);
...
if (ptr == (char*) (&SHELL_VERSION))
if (ptr == (char*) (&SHELL_VERSION))
```

Note. To make false alarms fewer, the V566 message is not generated for a range of cases. For instance, it does not appear if values -1, 0, 0xffffffff and 0xdeadbeef are magic numbers; if a number lies within the range between 0 and 65535 and is cast to a string pointer. This enables us to skip correct code fragments like the following one:

```
CString sMessage( LPCSTR IDS_FILE_WAS_CHANGED ) ;
This method of loading a string from resources is rather popular but certainly you'd better use MAKEINTRESOURCE. There are some other exceptions as well.
```

V567. Undefined behavior. The variable is modified while being used twice between sequence points.

The analyzer detected an expression leading to undefined behavior. A variable is used several times between two sequence points while its value is changing. We cannot predict the result of such an expression. Let's consider the notions "undefined behavior" and "sequence point" in detail.

Undefined behavior is a feature of some programming languages — most famously C/C++. In these languages, to simplify the specification and allow some flexibility in implementation, the specification leaves the results of certain operations specifically undefined.

For example, in C the use of any automatic variable before it has been initialized yields undefined behavior, as do division by zero and indexing an array outside of its defined bounds. This specifically frees the compiler to do whatever is easiest or most efficient, should such a program be submitted. In general, any behavior afterwards is also undefined. In particular, it is never required that the compiler diagnose undefined behavior — therefore, programs invoking undefined behavior may appear to compile and even run without errors at first, only to fail on another system, or even on another date. When an instance of undefined behavior occurs, so far as the language specification is concerned anything could happen, maybe nothing at all.

A sequence point in imperative programming defines any point in a computer program's execution at which it is guaranteed that all side effects of previous evaluations will have been performed, and no side effects from subsequent evaluations have yet been performed. They are often mentioned in reference to C and C++, because the result of some expressions can depend on the order of evaluation of their subexpressions. Adding one or more sequence points is one method of ensuring a consistent result, because this restricts the possible orders of evaluation.

Sequence points come into play when the same variable is modified more than once within a single expression. An often-cited example is the expression `i=i++`, which both assigns `i` to itself and increments `i`. The final value of `i` is ambiguous, because, depending on the language semantics, the increment may occur before, after or interleaved with the assignment. The definition of a particular language might specify one of the possible behaviors or simply say the behavior is undefined. In C and C++, evaluating such an expression yields undefined behavior.

C and C++ define the following sequence points:

1. Between evaluation of the left and right operands of the `&&` (logical AND), `||` (logical OR), and comma operators. For example, in the expression `*p++ != 0 && *q++ != 0`, all side effects of the sub-expression `*p++ != 0` are completed before any attempt to access `q`.
2. Between the evaluation of the first operand of the ternary "question-mark" operator and the second or third operand. For example, in the expression `a = (*p++) ? (*p++) : 0` there is a sequence point after the first `*p++`, meaning it has already been incremented by the time the second instance is executed.
3. At the end of a full expression. This category includes expression statements (such as the assignment `a=b;`), return statements, the controlling expressions of `if`, `switch`, `while`, or `do-while` statements, and all three expressions in a `for` statement.
4. Before a function is entered in a function call. The order in which the arguments are evaluated is not specified, but this sequence point means that all of their side effects are complete before the function is entered. In the expression `f(i++) + g(j++) + h(k++)`, `f` is called with a parameter of the original value of `i`, but `i` is incremented before entering the body of `f`. Similarly, `j` and `k` are updated before entering `g` and `h` respectively. However, it is not specified in which order `f()`, `g()`, `h()` are executed, nor in which order `i`, `j`, `k` are incremented. The values of `j` and `k` in the body of `f` are therefore undefined.[3] Note that a function call `f(a,b,c)` is not a use of the comma operator and the order of evaluation for `a`, `b`, and `c` is unspecified.
5. At a function return, after the return value is copied into the calling context. (This sequence point is only specified in the C++ standard; it is present only implicitly in C[4].)
6. At the end of an initializer; for example, after the evaluation of 5 in the declaration `int a = 5;`.
7. In C++, overloaded operators act as functions, so a call of an overloaded operator is a sequence point.

Now let's consider several samples causing undefined behavior:

```
int i, j;
...
X[i]=++i;
X[i++] = i;
j = i + X[++i];
i = 6 + i++ + 2000;
j = i++ + ++i;
i = ++i + ++i;
```

We cannot predict the calculation results in all these cases. Of course, these samples are artificial and we can notice the danger right away. So let's examine a code sample taken from a real application:

```
while (!(m_pBitArray[m_nCurrentBitIndex >> 5] &
        Powers_of_Two_Reversed[m_nCurrentBitIndex++ & 31]))
{
return (m_nCurrentBitIndex - BitInitial - 1);
}
```

The compiler can calculate either of the left or right arguments of the `'&'` operator first. It means that the `m_nCurrentBitIndex` variable might be already incremented by one when calculating `"m_pBitArray[m_nCurrentBitIndex >> 5]"`. Or it might still be not incremented.

This code may work well for a long time. However, you should keep in mind that it will behave correctly only when it is built in some particular compiler version with a fixed set of compilation options. This is the correct code:

```
while (!(m_pBitArray[m_nCurrentBitIndex >> 5] &
        Powers_of_Two_Reversed[m_nCurrentBitIndex & 31]))
{ ++m_nCurrentBitIndex; }
return (m_nCurrentBitIndex - BitInitial);
This code does not contain ambiguities anymore. We also got rid of the magic constant "-1".
```

Programmers often think that undefined behavior may occur only when using postincrement, while preincrement is safe. It's not so. Further is an example from a discussion on this subject.

Question:

I downloaded the trial version of your studio, ran it on my project and got this warning: V567 Undefined behavior. The `'i_acc'` variable is modified while being used twice between sequence points.

The code

```
i_acc = (++i_acc) % N_acc;
```

It seems to me that there is no undefined behavior because the `i_acc` variable does not participate in the expression twice.

Answer:

There is undefined behavior here. It's another thing that the probability of error occurrence is rather small in this case. The `'='` operator is not a sequence point. It means that the compiler might first put the value of the `i_acc` variable into the register and then increment the value in the register. After that it calculates the expression and writes the result into the `i_acc` variable and then again writes a register with the incremented value into the same variable. As a result we will get a code like this:

```
REG = i_acc;
REG++;
i_acc = (REG) % N_acc;
i_acc = REG;
```

The compiler has the absolute right to do so. Of course, in practice it will most likely increment the variable's value at once, and everything will be calculated as the programmer expects. But you should not rely on that.

References

1. Wikipedia. Undefined behavior. <http://www.viva64.com/go.php?url=668>
2. Wikipedia. Sequence point. <http://www.viva64.com/go.php?url=669>
3. Klaus Kreft & Angelika Langer. Sequence Points and Expression Evaluation in C++. <http://www.viva64.com/go.php?url=666>
4. Discussion at Bytes.com. Sequence points. <http://www.viva64.com/go.php?url=667>

V568. It's odd that the argument of sizeof() operator is the expression.

The analyzer detected a potential error: a suspicious expression serves as an argument of the sizeof() operator. Suspicious expressions can be arranged in two groups:

1. An expression attempts to change some variable.

The sizeof() operator calculates the expression's type and returns the size of this type. But the expression itself is not calculated. Here is a sample of suspicious code:

```
int A;
...
size_t size = sizeof(A++);
```

This code does not increment the 'A' variable. If you need to increment 'A', you'd better rewrite the code in the following way:

```
size_t size = sizeof(A);
A++;
```

2. Operations of addition, multiplication and the like are used in the expression.

Complex expressions signal errors. These errors are usually related to misprints. For example:

```
SendDlgItemMessage(
    hwndDlg, RULE_INPUT_1 + i, WM_GETTEXT,
    sizeof(buff - 1), (LPARAM) input_buff);
```

The programmer wrote "sizeof(buff - 1)" instead of "sizeof(buff) - 1". This is the correct code:

```
SendDlgItemMessage(
    hwndDlg, RULE_INPUT_1 + i, WM_GETTEXT,
    sizeof(buff) - 1, (LPARAM) input_buff);
```

Here is another sample of a misprint in program text:

```
memset(tcmtpt->stepsizes, 0,
    sizeof(tcmtpt->numstepsizes * sizeof(uint_fast16_t)));
```

The correct code:

```
memset(tcmtpt->stepsizes, 0,
    tcmtpt->numstepsizes * sizeof(uint_fast16_t));
```

V569. Truncation of constant value.

The analyzer detected a potential error: a constant value is truncated when it is assigned into a variable. Consider this sample:

```
int A[100];
unsigned char N = sizeof(A);
```

The size of the 'A' array (in Win32/Win64) is 400 bytes. The value range for unsigned char is 0..255. Consequently, the 'N' variable cannot store the size of the 'A' array.

The V569 warning tells you that you have chosen a wrong type to store this size or that you actually intended to calculate the number of items in the array instead of the array's size.

If you have chosen a wrong type, you may correct the code this way:

```
size_t N = sizeof(A);
```

If you intended to calculate the number of items in the array, you should rewrite the code this way:

```
unsigned char N = sizeof(A) / sizeof(*A);
```

V570. The variable is assigned to itself.

The analyzer detected a potential error: a variable is assigned to itself. Consider this sample:

```
dst.m_a = src.m_a;
dst.m_b = dst.m_b;
```

The value of the 'dst.m_b' variable will not change because of the misprint. This is the correct code:

```
dst.m_a = src.m_a;
dst.m_b = src.m_b;
```

The analyzer does not produce the warning every time it detects assignment of a variable to itself. This method is often used to suppress compiler-generated warnings. For example:

```
int Foo(int foo)
{
    UNREFERENCED_PARAMETER(foo);
    return 1;
}
```

The UNREFERENCED_PARAMETER macro is defined in the WinNT.h file in the following way:

```
#define UNREFERENCED_PARAMETER(P) \
{ \
    (P) = (P); \
}
```

However, the PVS-Studio analyzer knows about such cases and will not generate the V570 warning on assignment like this:

```
(foo) = (foo);
```

V571. Recurring check. This condition was already verified in previous line.

The analyzer detected a potential error: one and the same condition is checked twice. Consider two samples:

```
// Example N1:
if (A == B)
{
    if (A == B)
        ...
}

// Example N2:
if (A == B) {
} else {
    if (A == B)
        ...
}
```

In the first case, the second check "if (A==B)" is always true. In the second case, the second check is always false.

It is highly probable that this code has an error. For instance, a wrong variable name is used because of a misprint. This is the correct code:

```
// Example N1:
if (A == B)
{
    if (A == C)
        ...
}

// Example N2:
if (A == B) {
} else {
    if (A == C)
        ...
}
```

V572. It is odd that the object which was created using 'new' operator is immediately casted to another type.

The analyzer detected a potential error: an object created by the 'new' operator is explicitly cast to a different type. For example:

```
T_A *p = (T_A *) (new T_B());
...
delete p;
```

There are three possible ways of how this code has appeared and what to do with it.

1) T_B was not inherited from the T_A class.

Most probable, it is an unfortunate misprint or crude error. The way of correcting it depends upon the purpose of the code.

2) T_B is inherited from the T_A class. The T_A class does not have a virtual destructor.

In this case you cannot cast T_B to T_A because you will not be able to correctly destroy the created object then. This is the correct code:

```
T_B *p = new T_B();
...
delete p;
```

3) T_B is inherited from the T_A class. The T_A class has a virtual destructor.

In this case the code is correct but the explicit type conversion is meaningless. We can write it in a simpler way:

```
T_A *p = new T_B();
...
delete p;
```

There can be other cases when the V572 warning is generated. Let's consider a code sample taken from a real application:

```
DWORD CCompRemoteDriver::Open(HDRVR,
    char *, LPVIDEO_OPEN_PARAMS)
{
    return (DWORD) new CCompRemote();
}
```

The program handles the pointer as a descriptor for its purposes. To do that, it explicitly converts the pointer to the DWORD type. This code will work correctly in 32-bit systems but might fail in a 64-bit program. You may avoid the [64-bit error](#) using a more suitable data type DWORD_PTR:

```
DWORD_PTR CCompRemoteDriver::Open(HDRVR,
    char *, LPVIDEO_OPEN_PARAMS)
{
    return (DWORD_PTR) new CCompRemote();
}
```

Sometimes the V572 warning may be aroused by an atavism remaining since the time when the code was written in C. Let's consider such a sample:

```
struct Joint {
    ...
};
joints=(Joint*)new Joint[n]; //malloc(sizeof(Joint)*n);
```

The comment tells us that the 'malloc' function was used earlier to allocate memory. Now it is the 'new' operator which is used for this purpose. But the programmers forgot to remove the type conversion. The code is correct but the type conversion is needless here. We may write a shorter code:

```
joints = new Joint[n];
```

V573. Uninitialized variable 'Foo' was used. The variable was used to

initialize itself.

The analyzer detected a potential error: a variable being declared is used to initialize itself. Let's consider a simple synthetic sample:

```
int X = X + 1;
```

The X variable will be initialized by a random value. Of course, this sample is farfetched yet it is simple and good to show the warning's meaning. In practice, such an error might occur in more complex expressions. Consider this sample:

```
void Class::Foo(const std::string &FileName)
{
    if (FileName.empty())
        return;
    std::string FullName = m_Dir + std::string("\\") + FullName;
    ...
}
```

Because of the misprint in the expression, it is the FullName name which is used instead of FileName. This is the correct code:

```
std::string FullName = m_Dir + std::string("\\") + FileName;
```

V574. The pointer is used simultaneously as an array and as a pointer to single object.

The analyzer detected a potential error: a variable is used simultaneously as a pointer to a single object and as an array. Let's study a sample of the error the PVS-Studio analyzer has found in itself:

```
TypeInfo *factArgumentsTypeInfo =
    new (GC_QuickAlloc) TypeInfo[factArgumentsCount];
for (size_t i = 0; i != factArgumentsCount; ++i)
{
    Typeof(factArguments[i], factArgumentsTypeInfo[i]);
    factArgumentsTypeInfo->Normalize();
}
```

It is suspicious that we handle the factArgumentsTypeInfo variable as the "factArgumentsTypeInfo[i]" array and as a pointer to the single object "factArgumentsTypeInfo->". Actually we should call the Normalize() function for all the items. This is the fixed code:

```
TypeInfo *factArgumentsTypeInfo =
    new (GC_QuickAlloc) TypeInfo[factArgumentsCount];
for (size_t i = 0; i != factArgumentsCount; ++i)
{
    Typeof(factArguments[i], factArgumentsTypeInfo[i]);
    factArgumentsTypeInfo[i].Normalize();
}
```

V575. Function receives an odd argument.

The analyzer found a potential error: the function receives a very odd value as an actual argument.

Consider the sample:

```
bool Matrix4::operator==(const Matrix4& other) const {
    if (memcmp(this, &other, sizeof(Matrix4)) == 0)
        return true;
    ...
}
```

We deal with a misprint here: one round bracket is in a wrong place. Unfortunately, this error is not clearly visible and might exist in the code for a long time. Because of this misprint the size of memory being compared is calculated with the "sizeof(Matrix4) == 0" expression. Since the result of the expression is 'false', 0 bytes of memory are compared. This is the fixed code:

```
bool Matrix4::operator==(const Matrix4& other) const {
    if (memcmp(this, &other, sizeof(Matrix4)) == 0)
        return true;
    ...
}
```

V576. Incorrect format. Consider checking the N actual argument of the 'Foo' function.

The analyzer has detected a potential issue with the application of formatted output functions. (printf, sprintf, wprintf etc.) The formatting string doesn't correspond with actual arguments passed into the function. Let's review a simple example:

```
int A = 10;
double B = 20.0;
printf("%i %i\n", A, B);
```

According to the formatting string the 'printf' function is expecting two actual arguments of the 'int' type. However the second argument's value is of the 'double' type. Such an inconsistency leads to undefined behavior of a program. For example, it can lead to the output of senseless values.

The correct version:

```
int A = 10;
double B = 20.0;
printf("%i %f\n", A, B);
```

It's possible to cite countless examples of 'printf' function's incorrect use. Let's review some of the typical examples that are the most frequently encountered in applications.

Address printout.

The value of a pointer is quite commonly printed using these lines:

```
int *ptr = new int[100];
printf("0x%0.8X\n", ptr);
```

This source code is invalid because it will function properly only on systems which have their pointer size equal to size of 'int' type. For example In Win64 this code will print only the low-order part of the 'ptr' pointer. The correct version:

```
int *ptr = new int[100];
printf("0x%p\n", ptr);
```

The analyzer has detected the potential issue with an odd value being passed as the function's actual argument.

Unused arguments.

You can often encounter function calls in which some of these function's arguments are being unused.

For instance:

```
int nDOW;
#define KEY_ENABLED "Enabled"
...
wsprintf(cDowKey, "EnableDOW%d", nDOW, KEY_ENABLED);
```

It is obvious that the KEY_ENABLED parameter is unnecessary here or the source code should look like this:

```
wsprintf(cDowKey, "EnableDOW%d%s", nDOW, KEY_ENABLED);
```

Insufficient number of arguments.

A little more dangerous is the situation in which the number of arguments passed to the function is less than necessary. This can easily lead to the memory access error, buffer overflow or senseless printout. Let's review an example of memory allocation function taken from a real life application:

```
char* salloc(register int nbytes)
{
    register char* p;
    p = (char*) malloc((unsigned)nbytes);
    if (p == (char *)NULL)
    {
        fprintf(stderr, "%s: out of memory\n");
        exit(1);
    }
    return (p);
}
```

If 'malloc' returns NULL, the program will not be able to report the shortage of memory and to be terminated correctly. It instead will be terminated emergently and it will output the senseless text. In any case such a behavior will complicate analysis of the program's inoperability.

Confusion with signed/unsigned

Developers often employ the character printing specifier ('%i' for example) to output the variables of unsigned type. And vice versa. This error usually is not critical and is encountered so often than it has a low priority in PVS-Studio. In many cases such source code works flawlessly and fails only with large or negative values. Let us examine the code which is not correct, but successfully works:

```
int A = 10;
printf("A = %u\n", A);
for (unsigned i = 0; i != 5; ++i)
    printf("i = %d\n", i);
```

Although there is an inconsistency here, this code outputs correct values in practice. Of course it's better not to do this and to write correctly:

```
int A = 10;
printf("A = %d\n", A);
for (unsigned i = 0; i != 5; ++i)
    printf("i = %u\n", i);
```

The error will manifest itself in case there are large or negative values in the program. An Example:

```
int A = -1;
printf("A = %u", A);
```

Instead of "A=-1" string the program will print "A=4294967295". The correct version:

```
printf("A = %i", A);
```

Additional reference:

1. Wikipedia. Printf. <http://www.viva64.com/go.php?url=292>
2. MSDN. Format Specification Fields: printf and wprintf Functions. <http://www.viva64.com/go.php?url=693>

V577. Label is present inside a switch(). It is possible that these are misprints and 'default:' operator should be used instead.

The analyzer detected a potential error inside the switch operator. A label is used whose name is similar to 'default'. A misprint is probable. Consider this sample:

```
int c = 10;
int r = 0;
switch(c){
case 1:
    r = 3; break;
case 2:
    r = 7; break;
default:
    r = 8; break;
}
```

It seems that after the code's work is done, the value of the 'r' variable will be 8. Actually the 'r' variable will still equal zero. The point is that "defalt" is a label, not the "default" operator. This is the correct code:

```
int c = 10;
int r = 0;
switch(c){
case 1:
    r = 3; break;
case 2:
    r = 7; break;
default:
    r = 8; break;
}
```


V578. An odd bitwise operation detected. Consider verifying it.

The analyzer detected a potential error in an expression handling bits. A part of the expression is meaningless or excessive. Usually such errors occur due to a misprint. Consider this sample:

```
if (up & (PARAMETER_DPDU | PARAMETER_DPDU | PARAMETER_NG))
```

The PARAMETER_DPDU constant is used twice here. In a correct code there must be two different constants: PARAMETER_DPDU and PARAMETER_DPDV. The letter 'U' resembles 'V' and that is why this misprint has occurred. This is the correct code:

```
if (up & (PARAMETER_DPDU | PARAMETER_DPDV | PARAMETER_NG))
```

Another example. There is no error here but the code is excessive:

```
if ((pfds[i].dwFlags & pPFD->dwFlags) & pPFD->dwFlags)
    != pPFD->dwFlags)
```

This is a shorter code:

```
if ((pfds[i].dwFlags & pPFD->dwFlags) != pPFD->dwFlags)
```

V579. The 'Foo' function receives the pointer and its size as arguments. It is possibly a mistake. Inspect the N argument.

The analyzer detected an odd function call in code. A pointer and the size of the pointer are passed into a function as its arguments. Actually it is a common case when developers want to pass a buffer size instead of a pointer size into a function.

Let's see how an error like that can appear in code. Assume we had the following code in the beginning:

```
char buf[100];
...
memset(buf, 0, sizeof(buf));
```

The code is correct. The memset() function clears an array of 100 bytes. Then the code was changed and the buffer became variable-sized. The programmer forgot to change the code of buffer clearing:

```
char *buf = new char[N];
...
memset(buf, 0, sizeof(buf));
```

Now the code is incorrect. The sizeof() operator returns the pointer size instead of the size of the buffer with data. As a result, the memset() function clears only part of the array.

Let's consider another sample taken from a real application:

```
apr_size_t ap_regerror(int errcode,
    const ap_regex_t *preg, char *errbuf,
    apr_size_t errbuf_size)
{
    ...
    apr_snprintf(errbuf, sizeof errbuf,
        "%s%-6d", message, addmessage,
        (int)preg->re_erroffset);
    ...
}
```

It is not easy to notice the error in this code. The apr_snprintf() function accepts the 'errbuf' pointer and the size of this pointer 'sizeof errbuf' as arguments. The PVS-Studio analyzer considers this code odd and is absolutely right. The buffer size is stored in the 'errbuf_size' variable and it is this variable that should be used. This is the correct code:

```
apr_snprintf(errbuf, errbuf_size,
    "%s%-6d", message, addmessage,
    (int)preg->re_erroffset);
```

V580. An odd explicit type casting. Consider verifying it.

The analyzer detected an odd explicit type conversion. It may be either an error or a potential error.

Consider this sample:

```
DWORD errCode = 0;
void* dwErrParams[MAX_MESSAGE_PARAMS];
dwErrParams[0] = *((void*)&errCode);
```

The code contains a 64-bit error. The 'DWORD' type is cast to 'void *' type. This code works incorrectly in 64-bit systems where the pointer's size does not coincide with the size of the DWORD type. This is the correct code:

```
DWORD_PTR errCode = 0;
void* dwErrParams[MAX_MESSAGE_PARAMS];
dwErrParams[0] = (void *)errCode;
```

V581. The conditional expressions of the 'if' operators situated alongside each other are identical.

The analyzer detected code where there are two 'if' operators with identical close to each other. This is either a potential error or excessive code.

Consider the following sample:

```
if (strlen(S_1) == SIZE)
    Foo(A);
if (strlen(S_1) == SIZE)
    Foo(B);
```

Whether this code contains an error or not, depends upon what exactly the programmer intended to do. If the second condition must calculate the length of the other string, then it is an error. This is the correct code:

```
if (strlen(S_1) == SIZE)
```

```

    Foo(A);
if (strlen(S_2) == SIZE)
    Foo(B);

```

Maybe the code is correct, but it is inefficient in this case because it has to calculate the length of one and the same string twice. This is the optimized code:

```

if (strlen(S_1) == SIZE) {
    Foo(A);
    Foo(B);
}

```

V582. Consider reviewing the source code which operates the container.

The analyzer detected a potential error related to handling a fixed-sized container. One of our users advised us to implement this diagnostic. This is how he has formulated the task.

In order to handle arrays of a fixed size, we use the following template class:

```

template<class T_, int numElements> class idArray
{
public:
    int Num() const { return numElements; };
    ....
    inline const T_ & operator[](int index) const {
        idassert(index >= 0);
        idassert(index < numElements);
        return ptr[index];
    };
    inline T_ & operator[](int index) {
        idassert(index >= 0);
        idassert(index < numElements);
        return ptr[index];
    };
private:
    T_ ptr[numElements];
};

```

It has no performance overhead in release builds, but does index range checking in debug builds. Here is an example of incorrect code:

```

idArray<int, 1024> newArray;
newArray[-1] = 0;
newArray[1024] = 0;

```

The errors will be detected on launching the debug version. But we would like to be able to detect such errors using static analysis at the compilation time.

It is this type of issues that the V582 diagnostic is intended to detect. If a class is used in a program that makes use of a fixed-sized container's functionality, the analyzer tries to make sure that the index does not go beyond its boundaries. Here are examples of this diagnostic:

```

idArray<float, 16> ArrA;
idArray<float, 8> ArrB;
for (size_t i = 0; i != 16; i++)
    ArrA[i] = 1.0f;
for (size_t i = 0; i != 16; i++)
    ArrB[i] = 1.0f;

```

The analyzer will generate the following message on this code:

V582 Consider reviewing the source code which operates the 'ArrB' container. The value of the index belongs to the range: [0..15].

The error here is that the both loops handle 16 items, although the second array contains only 8 items. This is the correct code:

```

for (size_t i = 0; i != 16; i++)
    ArrA[i] = 1.0f;
for (size_t i = 0; i != 8; i++)
    ArrB[i] = 1.0f;

```

Note that passing of too large or too small indexes does not necessarily indicate an error in the program. For instance, the [] operator can be implemented in the following way:

```

inline T_ & operator[](int index) {
    if (index < 0) index = 0;
    if (index >= numElements) index = numElements - 1;
    return ptr[index];
};

```

If you use such classes and get too many false reports, you should turn off the V582 diagnostic.

Note. The analyzer does not possess an AI and its capabilities of searching for defects when handling containers are limited. We are working on improving the algorithms, so if you have noticed obviously false reports or, on the contrary, cases when the analyzer does not generate the warning, please write to us and send us the corresponding code sample.

V583. The '?' operator, regardless of its conditional expression, always returns one and the same value.

Analyzer found a potential error with utilization of "?:" ternary operator. Regardless of its conditional expression, the same operation will be performed. It is quite possible that a misprint is present in the source code.

Let's review the most basic example:

```

int A = B ? C : C;

```

In all cases the value of C variable will be assigned to the A variable.

Let's review how such a mistake could appear in the source code of real-life application:

```

fovRadius[0] =
    tan(DEG2RAD((rollAngleClamped % 2 == 0 ?
    cg.refdef.fov_x : cg.refdef.fov_x) * 0.52)) * sdist;

```

The code here is formatted. In the program's sources this is a single line of code and it is not surprising that the misprint could be overlooked quite easily. The

essence of an error is that the member of the "fov_x" structure is used twice.

The correct code:

```
fovRadius[0] =  
    tan(DEG2RAD((rollAngleClamped % 2 == 0 ?  
    cg.refdef.fov_x : cg.refdef.fov_y) * 0.52)) * sdist;
```

V584. The same value is present on both sides of the operator. The expression is incorrect or it can be simplified.

Analyzer found an expression that can be simplified. The possibility of a misprint presence in it is quite high. Let's review an example:

```
float SizeZ;  
if (SizeZ + 1 < SizeZ)
```

The analyzer thinks that this condition contains a mistake because it is practically senseless. Most likely another check was implied. The correct variant:

```
if (SizeZ + 1 < maxSizeZ)
```

Of course programmers sometimes utilize some tricks which are formally correct but do appear quite odd. The PVS-Studio analyzer tries to detect such situations if possible and not to produce warnings. For instance the analyzer considers such checks as being safe:

```
//overflow test for summation  
int a, b;  
if (a + b < a)  
  
//Verifying that x does not equals zero, +infinity, -infinity  
double X;  
if (X * 0.5f != X)
```

V585. An attempt to release the memory in which the 'Foo' local variable is stored.

Analyzer detected an attempt to release the memory occupied by the local variable. Such errors could be produced in case of careless refactoring or as misprints.

Let's review an example of the incorrect code:

```
void Foo()  
{  
    int *p;  
    ...  
    free(&p);  
}
```

The corrected code:

```
void Foo()  
{  
    int *p;  
    ...  
    free(p);  
}
```

V586. The 'Foo' function is called twice for deallocation of the same resource.

Analyzer detected a potential error of recurrent resource deallocation. The resource mentioned could be a memory space, some file or, for example, an HBRUSH object.

Let's review the example of incorrect code:

```
float *p1 = (float *)malloc(N * sizeof(float));  
float *p2 = (float *)malloc(K * sizeof(float));  
...  
free(p1);  
free(p1);
```

There is a misprint in application's source code which causes the double deallocation of same memory space. It is hard to predict the consequences of such code's execution. It's possible that such a program would crash. Or it will continue its execution but memory leak will occur.

The correct example:

```
float *p1 = (float *)malloc(N * sizeof(float));  
float *p2 = (float *)malloc(K * sizeof(float));  
...  
free(p1);  
free(p2);
```

Sometimes an error of double resource deallocation is not a dangerous one:

```
vector<unsigned> m_arrStack;  
...  
m_arrStack.clear();  
m_arrBlock.clear();  
m_arrStack.clear();
```

Accidentally the array is emptied twice. The code operates correctly but still it should be reviewed and corrected. During its study, it could be discovered that another array dissaallocation should have been performed nevertheless.

```
The correct example:  
vector<unsigned> m_arrStack;  
...  
m_arrStack.clear();  
m_arrBlock.clear();
```

V587. An odd sequence of assignments of this kind: A = B; B = A;.

Analyzer detected a potential error concerning the senseless mutual assignment of variables.

Let's review an example:

```
int A, B, C;
...
A = B;
C = 10;
B = A;
```

Here the assignment "B=A" lacks any sort of practical utility. It is possibly a misprint or just an unnecessary operation. The correct code:

```
A = B;
C = 10;
B = A_2;
```

An example stated above is a synthetic one. Let's see how such an error could appear in the source code of a real-life application:

```
// Swap; exercises counters
{
    RCPFooRef temp = f2;
    f2 = f3;
    f3 = f2;
}
```

The correct code:

```
// Swap; exercises counters
{
    RCPFooRef temp = f2;
    f2 = f3;
    f3 = temp;
}
```

V588. The expression of the 'A += B' kind is utilized. Consider reviewing it, as it is possible that 'A ++ B' was meant.

The analyzer detected a potential error: there is a sequence of '++' characters. It might be a misprint and you should use the '+= ' operator.

Consider the following example:

```
size_t size, delta;
...
size+=delta;
```

This code may be correct, but it is highly probable that there is a misprint and the programmer actually intended to use the '++' operator. This is the fixed code:

```
size_t size, delta;
...
size+=delta;
```

If this code is correct, you may remove '+' or type in an additional space to prevent showing the V588 warning. The following is an example of correct code where the warning is not generated:

```
size = delta;
size = +delta;
```

Note. To search for misprints of the 'A -= B' kind, we use the V589 diagnostic rule. This check is implemented separately since a lot of false reports are probable and you may want to disable it.

V589. The expression of the 'A -= B' kind is utilized. Consider reviewing it, as it is possible that 'A -- B' was meant.

The analyzer detected a potential error: there is a sequence of '--' characters in code. It might be a misprint and you should use the '-=' operator.

Consider this sample:

```
size_t size, delta;
...
size -= delta;
```

This code may be correct, but it is highly probable that there is a misprint and the programmer actually intended to use the '-=' operator. This is the fixed code:

```
size_t size, delta;
...
size -= delta;
```

If the code is correct, you may type in an additional space between the characters '=' and '-' to remove the V589 warning. This is an example of correct code where the warning is not generated:

```
size = -delta;
```

To make false reports fewer, there are some specific exceptions to the V589 rule. For instance, the analyzer will not generate the warning if a programmer does not use spaces between variables and operators. Here you are some samples of code the PVS-Studio analyzer considers safe:

```
A=-B;
int Z -= 1;
N -= N;
```

Note. To search for misprints of the 'A += B' type, the V588 diagnostic check is used.

V590. Consider inspecting this expression. The expression is excessive or contains a misprint.

The analyzer detected a potential error: there is an excessive comparison in code. Let me explain this by a simple example:

```
if (Aa == 10 && Aa != 3)
```

The condition will hold if 'Aa == 10'. The second part of the expression is meaningless. On studying the code, you may come to one of the two conclusions:

1) The expression can be simplified. This is the fixed code:

```
if (Aa == 10)
```

2) The expression has an error. This is the fixed code:

```
if (Aa == 10 && Bb != 3)
```

Here is an example of how this error may look in a real application:

```
int appliedSize, appliedSign;
...
if(appliedSize == 'b' && appliedSize != 's' && ...)
...
```

The expression has a misprint which is the reason why the appliedSize variable is used twice while appliedSign is not used at all. This is the fixed code:

```
int appliedSize, appliedSign;
...
if(appliedSize == 'b' && appliedSign != 's' && ...)
...
```

Let's study another example from practice. We have no error here, but the expression is excessive, which might make the code less readable:

```
while (*pBuff == ' ' && *pBuff != '\0')
    pBuff++;
```

The " *pBuff != '\0' " check is meaningless. This is the shortened code:

```
while (*pBuff == ' ')
    pBuff++;
```

V591. Non-void function should return a value.

The analyzer detected a function that returns a random value. It might be an error.

Consider this sample:

```
int main (int argc, char** argv)
{
    ...
    printf("FINISH\r\n");
}
```

The main() function returns an integer number which is accepted by the calling process. If main() does not return a value explicitly, the calling process gets a nominally undefined value. This is the correct code:

```
int main (int argc, char** argv)
{
    ...
    printf("FINISH\r\n");
    return retCode;
}
```

A more interesting and dangerous case is when we deal with code of functions where an undefined value is returned only sometimes. Consider the following sample:

```
BOOL IsInterestingString(char *s)
{
    if (s == NULL)
        return FALSE;
    if (strlen(s) < 4)
        return;
    return (s[0] == '#') ? TRUE : FALSE;
}
```

There is a misprint in the code. If a string's length is less than 4 characters, the function will return an undefined value. This is the correct code:

```
BOOL IsInterestingString(char *s)
{
    if (s == NULL)
        return FALSE;
    if (strlen(s) < 4)
        return FALSE;
    return (s[0] == '#') ? TRUE : FALSE;
}
```

Note. The analyzer tries to determine cases when absence of a returned value is not an error. Here is an example of code PVS-Studio will consider safe:

```
int Foo()
{
    ...
    exit(10);
}
```

V592. The expression was enclosed by parentheses twice: ((expression)). One pair of parentheses is unnecessary or misprint is present.

The analyzer detected double parentheses enclosing an expression. It is probable that one of the brackets is in a wrong place.

Note that the analyzer does not search for code fragments where parentheses are used twice. For instance, the analyzer considers the check "if ((A = B))" safe. Additional parentheses are used here to suppress warnings of some compilers. You cannot arrange parentheses in this expression so that an error occurs.

The PVS-Studio analyzer tries to find cases when you may change an expression's meaning by changing a location of one bracket. Consider the following

sample:

```
if(!(!osx||howmanylevels))
```

This code is suspicious. The purpose of additional parentheses here is not clear. Perhaps the expression should look this way:

```
if(!(osx||howmanylevels))
```

Even if the expression is correct, we still should remove the additional parentheses. There are two reasons for that.

- 1) A person reading the code may doubt that it is correct on seeing double parentheses.
- 2) If you remove additional parentheses, the analyzer will stop generating a false report.

V593. Consider reviewing the expression of the 'A = B == C' kind. The expression is calculated as following: 'A = (B == C)'.

The analyzer detected a potential error in an expression that is most probably working in a way other than intended by the programmer. Most often you may see errors of this type in expressions where an assignment operation and operation of checking a function's result are performed simultaneously. Consider a simple example:

```
if (handle = Foo() != -1)
```

While creating this code, the programmer usually wants the actions to be performed in the following order:

```
if ((handle = Foo()) != -1)
```

But the priority of the '!=' operator is higher than that of the '=' operator. That is why the expression will be calculated in the following way:

```
if (handle = (Foo() != -1))
```

To fix the error, you may use parentheses, or rather not be stingy with code lines. Your program's text will become more readable if you write it this way:

```
handle = Foo();  
if (handle != -1)
```

Let's see how such an error might look in a real application:

```
if (hr = AVIFileGetStream(pfileSilence,  
    &paviSilence, typeAUDIO, 0) != AVIERR_OK)  
{  
    ErrMsg("Unable to load silence stream");  
    return hr;  
}
```

The check in the code where the error has occurred works correctly and we will get the message "Unable to load silence stream". The trouble is that the 'hr' variable will store value 1 and not the error's code. This is the fixed code:

```
if ((hr = AVIFileGetStream(pfileSilence,  
    &paviSilence, typeAUDIO, 0)) != AVIERR_OK)  
{  
    ErrMsg("Unable to load silence stream");  
    return hr;  
}
```

The analyzer does not always generate warnings on detecting a construct of the "if (x = a == b)" kind. For instance, the analyzer understands that the following code is safe:

```
char *from;  
char *to;  
bool result;  
...  
if (result = from == to)  
{}
```

Note. If the analyzer still generates a false alarm, you may use two methods to suppress it:

- 1) Add one more pair of parentheses. For example: "if (x = (a == b))".
- 2) Use a comment to suppress the warning. For example: "if (x = a == b) //-V593".

V594. The pointer steps out of array's bounds.

The analyzer has detected a potential error of pointer handling. There is an expression in the program, on calculating which a pointer leaves array bounds. Here is a simple example to clarify this point:

```
int A[10];  
fill(A, A + sizeof(A), 33);
```

We want all the array items to be assigned value 33. The error is this: the "A + sizeof(A)" pointer points far outside the array's bounds. As a result, we will change more memory cells than intended. A result of such an error is unpredictable.

This is the correct code:

```
int A[10];  
fill(A, A + sizeof(A) / sizeof(A[0]), 33);
```

V595. The pointer was utilized before it was verified against nullptr. Check lines: N1, N2.

The analyzer has detected a potential error that may cause dereferencing of a null pointer.

The analyzer has noticed the following situation in the code: a pointer is being used first and only then it is checked whether or not this is a NULL pointer. It means one of the two things:

- 1) An error occurs if the pointer is equal to NULL.

2) The program works correctly, since the pointer is never equal to NULL. The check is not necessary in this case.

Let's consider the first case. There is an error.

```
buf = Foo();
pos = buf->pos;
if (!buf) return -1;
```

If the 'buf' pointer is equal to NULL, the 'buf->pos' expression will cause an error. The analyzer will generate a warning for this code mentioning 2 lines: the first line is the place where the pointer is used; the second line is the place where the pointer is compared to NULL.

This is the correct code:

```
buf = Foo();
if (!buf) return -1;
pos = buf->pos;

pos = buf->pos;
```

Let's consider the second case. There is no error.

```
void F(MyClass *p)
{
    if (!IsOkPtr(p))
        return;
    printf("%s", p->Foo());
    if (p) p->Clear();
}
```

This code is always correct. The pointer is never equal to NULL. But the analyzer does not understand this situation and generates a warning. To make it disappear, you should remove the check "if (p)". It has no sense and can only confuse a programmer reading this code.

This is the correct code:

```
void F(MyClass *p)
{
    if (!IsOkPtr(p))
        return;
    printf("%s", p->Foo());
    p->Clear();
}
```

When the PVS-Studio analyzer is mistaken, you may use (apart from changing the code) a comment to suppress warnings. For example: "p->Foo(); //-V595".

V596. The object was created but it is not being used. The 'throw' keyword could be missing.

The analyzer has detected a strange use of the std::exception class or derived class. The analyzer generates this warning when an object of the std::exception / CException type is created but not being used. For example:

```
if (name.empty())
    std::logic_error("Name mustn't be empty");
```

The error is this: the key word 'throw' is missing by accident. As a result, this code does not generate an exception in case of an error. This is the fixed code:

```
if (name.empty())
    throw std::logic_error("Name mustn't be empty");
```

V597. The compiler could delete the 'memset' function call, which is used to flush 'Foo' buffer. The RtlSecureZeroMemory() function should be used to erase the private data.

The analyzer has detected a potential error: an array containing private information is not cleared.

Consider the following code sample.

```
void Foo()
{
    char password[MAX_PASSWORD_LEN];
    InputPassword(password);
    ProcessPassword(password);
    memset(password, 0, sizeof(password));
}
```

The function on the stack creates a temporary buffer intended for password storage. When we finish working with the password, we want to clear this buffer. If you don't do this, the password will remain in memory, which might lead to unpleasant consequences. Article about this: "[Overwriting memory - why?](#)".

Unfortunately, the code above may leave the buffer uncleared. Note that the 'password' array is cleared at the end and is not used anymore. That's why when building the Release version of the application, the compiler will most likely delete the call of the memset() function. The compiler has an absolute right to do that. This change does not affect the observed behavior which is described in the Standard as a sequence of calls of input-output functions and volatile data read-write functions. That is, from the viewpoint of the C/C++ language removing the call of the memset() function does not change anything!

To clear buffers containing private information you should use a special function [RtlSecureZeroMemory\(\)](#). This is the fixed code:

```
void Foo()
{
    char password[MAX_PASSWORD_LEN];
    InputPassword(password);
    ProcessPassword(password);
    RtlSecureZeroMemory(password, sizeof(password));
}
```

It seems that in practice the compiler cannot delete a call of such an important function as memset(). You might think that we speak of some exotic compilers. It's not so. Take the Visual C++ 10 compiler included into Visual Studio 2010, for instance.

Let's consider the two functions.

```
void Fl()
{
```



```

TCHAR buf[100];
_sprintf(buf, _T("Test: %d"), 123);
MessageBox(NULL, buf, NULL, MB_OK);
memset(buf, 0, sizeof(buf));
}

void F2()
{
TCHAR buf[100];
_sprintf(buf, _T("Test: %d"), 123);
MessageBox(NULL, buf, NULL, MB_OK);
RtlSecureZeroMemory(buf, sizeof(buf));
}

```

The functions differ in the way they clear the buffer. The first one uses the `memset()` function, and the second the `RtlSecureZeroMemory()` function. Let's compile the optimized code enabling the "/O2" switch for the Visual C++ 10 compiler. Look at the assembler code we've got as a result:

```

0000000013F71143 mov     r8d,7Bh
0000000013F7114B call    qword ptr [__imp__sprintf
    MessageBox(NULL, buf, NULL, MB_OK);
0000000013F71151 lea     rdx,[rsp+20h]
0000000013F71156 xor     r9d,r9d
0000000013F71159 xor     r8d,r8d
0000000013F7115C xor     ecx,ecx
0000000013F7115E call    qword ptr [__imp_MessageBo
    memset(buf, 0, sizeof(buf));
}
0000000013F71164 mov     rcx,qword ptr [rsp+0F0h]
0000000013F7116C xor     rcx,rcx
0000000013F7116F call    __security_check_cookie (1
0000000013F71174 add     rsp,100h
0000000013F7117B ret

```

Figure 1. The `memset()` function is removed.

```

0000000013F2511AD call    qword ptr [__imp__sprintf
    MessageBox(NULL, buf, NULL, MB_OK);
0000000013F2511B3 lea     rdx,[rsp+20h]
0000000013F2511B8 xor     r9d,r9d
0000000013F2511BB xor     r8d,r8d
0000000013F2511BE xor     ecx,ecx
0000000013F2511C0 call    qword ptr [__imp_MessageBo
    RtlSecureZeroMemory(buf, sizeof(buf));
0000000013F2511C6 lea     rdi,[rsp+20h]
0000000013F2511CB xor     eax,eax
0000000013F2511CD mov     ecx,0C8h
0000000013F2511D2 rep stos byte ptr [rdi]
}
0000000013F2511D4 mov     rcx,qword ptr [rsp+0F0h]
0000000013F2511DC xor     rcx,rcx
0000000013F2511DF call    __security_check_cookie (1
0000000013F2511E4 add     rsp,100h
0000000013F2511EB pop     rdi
0000000013F2511EC ret

```

Figure 2. The `RtlSecureZeroMemory()` function fills memory with nulls.

As you can see from the assembler code, the `memset()` function was deleted by the compiler during optimization, while the `RtlSecureZeroMemory()` function was arranged into the code, thus clearing the array successfully.

V598. The 'memset/memcpy' function is used to nullify/copy the fields of 'Foo' class. Virtual method table will be damaged by this.

The analyzer has detected that such low-level functions as `memset()` or `memcpy()` are used to handle a class. It is inadmissible when a class has a [virtual method table](#). The `memset()/memcpy()` functions might clear the table contents, and the program behavior will become undefined.

Consider the following code sample.

```

class MyClass
{
    int A, B, C;
    char buf[100];
    MyClass();
    virtual ~MyClass() {}
};

MyClass::MyClass()
{
    memset(this, 0, sizeof(*this));
}

```

Note that there is a virtual destructor in the class. It means that the class has a virtual method table. The programmer was too lazy to clear the class members separately and used the `memset()` function for that purpose. It will spoil the table, since the `memset()` function does not know anything about it.

This is the correct code:

```

MyClass::MyClass() : A(0), B(0), C(0)
{
    memset(buf, 0, sizeof(buf));
}

```

V599. The virtual destructor is not present, although the 'Foo' class contains virtual functions.

The analyzer has found a potential error: a virtual destructor is absent in a class. The following conditions must hold for the analyzer to generate the V599 warning:

- 1) A class object is destroyed by the delete operator.
- 2) The class has at least one virtual function.

Presence of virtual functions indicates that the class may be used polymorphically. In this case a virtual destructor is necessary to correctly destroy the object.

Consider the following code sample.

```
class Father
{
public:
    Father() {}
    ~Father() {}
    virtual void Foo() { ... }
};

class Son : public Father
{
public:
    int* buffer;
    Son() : Father() { buffer = new int[1024]; }
    ~Son() { delete[] buffer; }
    virtual void Foo() { ... }
};

...
Father* object = new Son();
delete object;           // Call ~Father()!!
```

The code is incorrect and leads to memory leak. At the moment of deleting the object only the destructor in the 'Father' class is called. To call the 'Son' class' destructor you should make the destructor virtual.

This is the correct code:

```
class Father
{
public:
    Father() {}
    virtual ~Father() {}
    virtual void Foo() { ... }
};
```

The V599 diagnostic message helps to detect far not all the issues related to absence of virtual destructors. Here is the corresponding example:

You develop a library. It contains the XXX class which has virtual functions but no virtual destructor. You don't handle this class in the library yourself, so the PVS-Studio analyzer won't warn you about the danger. The problem might occur at the side of a programmer who uses your library and whose classes are inheritance of the XXX class.

The [C4265: 'class': class has virtual functions, but destructor is not virtual](#) diagnostic message implemented in Visual C++ allows you to detect much more issues. This is a very useful message. But it is turned off by default. I cannot say why. This subject was discussed on the StackOverflow site: [Why is C4265 Visual C++ warning \(virtual member function and no virtual destructor\) off by default?](#) Unfortunately, nobody managed to give a reasonable explanation.

We suppose that C4265 gives many false positives in code where the [mixin](#) pattern is used. When using this pattern, a lot of interface classes appear which contain virtual functions but they don't need a virtual destructor.

We can say that the V599 diagnostic rule is a special case of C4265. It produces fewer false reports but, unfortunately, allows you to detect fewer defects. If you want to analyze your code more thoroughly, turn on the C4265 warning.

P. S.

Unfortunately, ALWAYS declaring a destructor as a virtual one is not a good programming practice. It leads to additional overhead costs, since the class has to store a pointer to the Virtual Method Table.

Additional resources:

1. Wikipedia. Virtual method table. <http://www.viva64.com/go.php?url=771>
2. Wikipedia. Virtual function. <http://www.viva64.com/go.php?url=780>
3. Wikipedia. Destructor. <http://www.viva64.com/go.php?url=781>
4. Discussion on StackOverflow. When to use virtual destructors? <http://www.viva64.com/go.php?url=782>
5. The Old New Thing. When should your destructor be virtual? <http://www.viva64.com/go.php?url=783>

V600. Consider inspecting the condition. The 'Foo' pointer is always not equal to NULL.

The analyzer has detected a comparison of an array address to null. This comparison is meaningless and might signal an error in the program.

Consider the following code sample.

```
void Foo()
{
    short T_IND[8][13];
    ...
    if (T_IND[1][j]==0 && T_IND[5]!=0)
        T_buf[top[0]] = top[1]*T_IND[6][j];
    ...
}
```

The program handles a two-dimensional array. The code is difficult to read, so the error is not visible at first sight. But the analyzer will warn you that the "T_IND[5]!=0" comparison is meaningless. The pointer "T_IND[5]" is always not equal to zero.

After studying the V600 warnings you may find errors which are usually caused by misprints. For instance, it may turn out that the code above should be written in the following way:

```
if (T_IND[1][j]==0 && T_IND[5][j]!=0)
```

The V600 warning doesn't always indicate a real error. Careless refactoring is often the reason for generating the V600 warning. Let's examine the most common case. This is how the code looked at first:

```
int *p = (int *)malloc(sizeof(int) *ARRAY_SIZE);

...

if (!p)

return false;

...

free(p);
```

Then it underwent some changes. It appeared that the ARRAY_SIZE value was small and the array was able to be created on the stack. As a result, we have the following code:

```
int p[ARRAY_SIZE];

...

if (!p)

return false;

...
```

The V600 warning is generated here. But the code is correct. It simply turns out that the "if (!p)" check has become meaningless and can be removed.

V601. An odd implicit type casting.

The analyzer has detected an odd implicit type conversion. Such a type conversion might signal an error or carelessly written code.

Let's consider the first example.

```
std::string str;
bool bstr;
...
str = false;
```

Any programmer will be surprised on seeing an assignment of the 'false' value to a variable of the 'std::string' type. But this construct is quite permissible and working. The 'false' value will turn into a null pointer. Then the 'str' variable will be assigned an empty string.

However, this is not what the programmer must have intended. If he/she wanted to clear the string, he/she would use the 'clear()' function for that. The programmer just made a mistake here and wrote a wrong variable.

This is the correct code:

```
std::string str;
bool bstr;
...
bstr = false;
```

Consider the second example:

```
bool Ret(int *p)
{
    if (!p)
        return "p1";
    ...
}
```

The string literal "p1" turns into the 'true' variable and is returned from the function. It is a very odd code.

We cannot give you general recommendations on fixing such code, since every case must be considered individually.

V602. Consider inspecting this expression. '<' possibly should be replaced with '<<'.

The analyzer has detected a potential error that may be caused by a misprint. It is highly probable that the '<<' operator must be used instead of '<' in an expression.

Consider the following code sample.

```
void Foo(unsigned nXNegYNegZNeg, unsigned nXNegYNegZPos,
         unsigned nXNegYPosZNeg, unsigned nXNegYPosZPos)
{
    unsigned m_nIVSampleDirBitmask =
        (1 << nXNegYNegZNeg) | (1 < nXNegYNegZPos) |
        (1 << nXNegYPosZNeg) | (1 << nXNegYPosZPos);
    ...
}
```

The code contains an error, since it is the '<' operator that is written by accident in the expression. This is the correct code:

```
unsigned m_nIVSampleDirBitmask =
    (1 << nXNegYNegZNeg) | (1 << nXNegYNegZPos) |
    (1 << nXNegYPosZNeg) | (1 << nXNegYPosZPos);
```

Note.

The analyzer considers comparisons ('<', '>') odd if their result is used in binary operations such as '&', '|' or '^'. The diagnostic is more complex but we hope you understand the point in general. On finding such expressions the analyzer emits the V602 warning.

If the analyzer produces a false positive error, you may suppress it using the "//V602" comment. But in most cases you'd better rewrite this code. It's not a good practice to handle expressions of the 'bool' type using binary operators: it makes the code uneventful and less readable.

V603. The object was created but it is not being used. If you wish to call constructor, 'this->Foo::Foo(...)' should be used.

The analyzer has detected a potential error: incorrect use of a constructor. Programmers often make mistakes trying to call a constructor explicitly to initialize an object.

Consider a typical sample taken from a real application:

```
class CSlideBarGroup
{
public:
    CSlideBarGroup(CString strName, INT iIconIndex,
                  CListBoxST* pListBox);
    CSlideBarGroup(CSlideBarGroup& Group);
    ...
};

CSlideBarGroup::CSlideBarGroup(CSlideBarGroup& Group)
{
    CSlideBarGroup(Group.GetName(), Group.GetIconIndex(),
                  Group.GetListBox());
}
```

There are two constructors in the class. To reduce the source code's size the programmer decided to call one constructor from the other. But this code does quite the other thing than intended.

The following happens: a new unnamed object of the CSlideBarGroup type is created and gets destroyed right after. As a result, the class fields remain uninitialized.

The correct way is to create an initialization function and call it from the constructors. This is the correct code:

```
class CSlideBarGroup
{
    void Init(CString strName, INT iIconIndex,
             CListBoxST* pListBox);
public:
    CSlideBarGroup(CString strName, INT iIconIndex,
                  CListBoxST* pListBox)
    {
        Init(strName, iIconIndex, pListBox);
    }
    CSlideBarGroup(CSlideBarGroup& Group)
    {
        Init(Group.GetName(), Group.GetIconIndex(),
              Group.GetListBox());
    }
    ...
};
```

If you still want to call the constructor, you may write it in this way:

```
CSlideBarGroup::CSlideBarGroup(CSlideBarGroup& Group)
{
    this->CSlideBarGroup::CSlideBarGroup(
        Group.GetName(), Group.GetIconIndex(), Group.GetListBox());
}
```

Another identical code:

```
CSlideBarGroup::CSlideBarGroup(CSlideBarGroup& Group)
{
    new (this) CSlideBarGroup(
        Group.GetName(), Group.GetIconIndex(),
        Group.GetListBox());
}
```

The code of the given samples is very dangerous and you should understand well how they work!

You may do more harm than good with this code. Consider the following samples showing where such a constructor call is admissible and where it is not.

```
class SomeClass
{
    int x,y;
public:
    SomeClass() { SomeClass(0,0); }
    SomeClass(int xx, int yy) : x(xx), y(yy) {}
};
```

The code contains an error. In the 'SomeClass()' constructor, a temporary object is created. As a result, the 'x' and 'y' fields remain uninitialized. You can fix the code in this way:

```
class SomeClass
{
    int x,y;
public:
    SomeClass() { new (this) SomeClass(0,0); }
    SomeClass(int xx, int yy) : x(xx), y(yy) {}
};
```

This code will work well. It is safe and working because the class contains primary data types and is not a descendant of other classes. In this case the double constructor call is not harmful.

Consider another code where the explicit constructor call causes an error:

```
class Base
{
public:
    char *ptr;
    std::vector vect;
    Base() { ptr = new char[1000]; }
    ~Base() { delete [] ptr; }
};

class Derived : Base
{
    Derived(Foo foo) { }
```

```

Derived(Bar bar) {
    new (this) Derived(bar.foo);
}
}

```

When we call the "new (this) Derived(bar.foo);" constructor, the Base object is already created and the fields are initialized. The repeated constructor call will lead to double initialization; we will write a pointer to the newly allocated memory area into 'ptr'. As a result we will get memory leak. And if you take double initialization of an object of the std::vector type, you cannot predict its result at all. But one thing is obvious: this code is inadmissible.

In conclusion, I want to note it once again that you'd better create an initialization function instead of explicitly calling a constructor. Explicit constructor call is needed only in very rare cases.

Explicit call of one constructor from the other in C++11 (delegation)

The new standard allows you to perform call of constructors from other constructors (known as delegation). It enables you to create constructors that use behavior of other constructors without added code.

This is an example of correct code:

```

class MyClass {
    int m_x;
public:
    MyClass(int X) : m_x(X) {}
    MyClass() : MyClass(33) {}
};

```

The MyClass constructor without arguments calls a constructor of the same class with an integer argument.

C++03 considers an object to be constructed when its constructor finishes executing, but C++11 considers an object constructed once any constructor finishes execution. Since multiple constructors will be allowed to execute, this will mean that each delegate constructor will be executing on a fully constructed object of its own type. Derived class constructors will execute after all delegation in their base classes is complete.

Additional information

1. Discussion on StackOverflow. C++'s "placement new". <http://www.viva64.com/go.php?url=791>
2. Discussion on StackOverflow. Using new (this) to reuse constructors. <http://www.viva64.com/go.php?url=790>

V801. Decreased performance. It is better to redefine the N function argument as a reference. Consider replacing 'const T' with 'const .. &T' / 'const .. *T'.

The analyzer detected a construct that can be optimized. An object of type class or structure is passed to a function. This object is passed by value but is not modified because there is the key word const. Perhaps you should pass this object using a constant reference in the C++ language or a pointer in the C language.

For example:

```

bool IsA(const std::string s)
{
    return s == A;
}

```

When calling this function, the copy constructor will be called for the std::string class. If objects are often copied this way, this may significantly reduce the application's performance. You may easily optimize the code by adding the reference:

```

bool IsA(const std::string &s)
{
    return s == A;
}

```

References:

1. Wikipedia. Reference (C++). <http://www.viva64.com/go.php?url=463>
2. Bjarne Stroustrup. The C++ Programming Language (Third Edition and Special Edition). 11.6 - Large Objects. <http://www.viva64.com/go.php?url=464>

V802. On 32-bit/64-bit platform, structure size can be reduced from N to K bytes by rearranging the fields according to their sizes in decreasing order.

The analyzer detected a construct which can be optimized. There is a data structure in program code that might cause inefficient use of memory.

Let's examine a sample of such a structure the analyzer considers inefficient:

```

struct LiseElement {
    bool m_isActive;
    char *m_pNext;
    int m_value;
};

```

This structure occupies 24 bytes in 64-bit code because of data alignment. But if you change the field sequence, its size will be only 16 bytes. This is the optimized structure:

```

struct LiseElement {
    char *m_pNext;
    int m_value;
    bool m_isActive;
};

```

Of course, field rearrangement is not always possible or necessary. But if you use millions of such structures, it is reasonable to optimize memory being consumed. Additional reduction of structures' sizes may increase the application's performance because fewer memory accesses will be needed at the same number of items.

Note that the structure described above always occupies 12 bytes in a 32-bit program regardless of the field sequence. That is why the V802 message will not be shown when checking the 32-bit configuration.

Surely there might be opposite cases when you can optimize a structure's size in the 32-bit configuration and cannot do that in the 64-bit configuration. Here is a sample of such a structure:

```
struct T_2
{
    int *m_p1;
    __int64 m_x;
    int *m_p2;
}
```

This structure occupies 24 bytes in the 32-bit program because of the alignment. If we rearrange the fields as shown below, its size will be only 16 bytes.

```
struct T_2
{
    __int64 m_x;
    int *m_p1;
    int *m_p2;
}
```

It does not matter how fields are arranged in the 'T_2' structure in the 64-bit configuration: it will occupy 24 bytes anyway.

The method of reducing structures' sizes is rather simple. You just need to arrange fields in descending order of their sizes. In this case, fields will be arranged without unnecessary gaps. For instance, take this structure of 40 bytes in a 64-bit program:

```
struct MyStruct
{
    int m_int;
    size_t m_size_t;
    short m_short;
    void *m_ptr;
    char m_char;
};
```

By simply sorting the sequence of fields in descending order of their sizes:

```
struct MyStructOpt
{
    void *m_ptr;
    size_t m_size_t;
    int m_int;
    short m_short;
    char m_char;
};
```

we get a structure with the size of 24 bytes.

The analyzer does not always generate messages about inefficient structures because it tries to make unnecessary warnings fewer. For instance, the analyzer does not generate this warning for complex descendant classes since there are usually rather few of such objects. For example:

```
class MyWindow : public CWnd {
    bool m_isActive;
    size_t m_sizeX, m_sizeY;
    char m_color[3];
    ...
};
```

This structure's size may be reduced but it does not give your practical benefit.

V803. Decreased performance. It is more effective to use the prefix form of ++it. Replace iterator++ with ++iterator.

The analyzer detected a construct which may be optimized. An iterator is changed in the program code by the increment/decrement postfix operator. Since the previous iterator's value is not used, you may replace the postfix operator with the prefix one. In some cases, the prefix operator will work faster than the postfix one, especially in Debug-versions.

Example:

```
std::vector<size_t>::const_iterator it;
for (it = a.begin(); it != a.end(); it++)
{ ... }
```

This code is faster:

```
std::vector<size_t>::const_iterator it;
for (it = a.begin(); it != a.end(); ++it)
{ ... }
```

The prefix increment operator changes the object's state and returns itself already changed. The prefix operator in the iterator's class to handle std::vector might look as follows:

```
_Myt& operator++()
{ // preincrement
    ++_Myptr;
    return (*this);
}
```

The situation with the postfix increment operator is more complicated. The object's state must change but it is the previous state which is returned. So an additional temporary object is created:

```
_Myt operator++(int)
{ // postincrement
    _Myt _Tmp = *this;
    ++*this;
    return (_Tmp);
}
```

If we want only to increment the iterator's value, it appears that the prefix version is preferable. So here you are one of the tips on micro-optimization of software: write "for (it = a.begin(); it != a.end(); ++it)" instead of "for (it = a.begin(); it != a.end(); it++)". In the latter case, an unnecessary temporary object is created, which reduces performance.

To study all these questions in detail, refer to the book by Scott Meyers "Efficient use of C++. 35 new recommendations on improving your programs and projects" (Rule 6. Distinguish between prefix increment and decrement operators) [1].

You may also study the results of speed measurements in the post "Is it reasonable to use the prefix increment operator ++it instead of postfix operator it++ for

iterators?" [2].

References

1. Meyers, Scott. More Effective C++: 35 New Ways to Improve Your Programs and Designs. Addison-Wesley, Reading, Mass., 1996. ISBN-10: 020163371X. ISBN-13: 9780201633719.
2. Andrey Karpov. Is it reasonable to use the prefix increment operator ++it instead of postfix operator it++ for iterators? <http://www.viva64.com/en/b/0093/>

V804. Decreased performance. The 'Foo' function is called twice in the specified expression to calculate length of the same string.

The analyzer detected a construct which can be potentially optimized. Length of one and the same string is calculated twice in one expression. For length calculation such functions as `strlen`, `lstrlen`, `_mbslen`, etc. are used. If this expression is calculated many times or strings have large lengths, this code fragment should be optimized.

For optimization purposes, you may preliminary calculate the string length and place it into a temporary variable.

For example:

```
if ((strlen(directory) > 0) &&
    (directory[strlen(directory)-1] != '\\'))
```

Most likely, this code processes only one string and it does not need optimization. But if the code is called very often, we should rewrite it. This is a better version of the code:

```
size_t directoryLen = strlen(directory);
if ((directoryLen > 0) && (directory[directoryLen-1] != '\\'))
```

Sometimes the V804 warning helps to detect much more crucial errors. Consider this sample:

```
if (strlen(str_1) > 4 && strlen(str_1) > 8)
```

An incorrect variable name is used here. This is the correct code:

```
if (strlen(str_1) > 4 && strlen(str_2) > 8)
```

V805. Decreased performance. It is inefficient to identify an empty string by using 'strlen(str) > 0' construct. A more efficient way is to check: `str[0] != '\0'`

The analyzer detected a construct that can be optimized. To determine whether a code string is empty or not, the `strlen` function or some other identical function is used. For example:

```
if (strlen(strUrl) > 0)
```

This code is correct, but if it is used inside a long loop or if we handle long strings, such a check might be inefficient. To check if a string is empty or not, we just have to compare the first character of the string with 0. This is an optimized code:

```
if (strUrl[0] != '\0')
```

Sometimes the V805 warning helps to detect excessive code. In one application we have found a code fragment like the following one:

```
string path;
...
if (strlen(path.c_str()) != 0)
if (strlen(path.c_str()) != 0)
```

Most likely, this code appeared during careless refactoring when the type of the path variable had been changed from a simple pointer to `std::string`. This is a shorter and faster code:

```
if (!path.empty())
```

V806. Decreased performance. The expression of `strlen(MyStr.c_str())` kind can be rewritten as `MyStr.length()`.

Analyzer found a construct which potentially can be optimized. The length of a string located in the container is calculated by using the `strlen()` function or by function similar to it. This operation is excessive, as the container possesses a special function for string length calculation.

Let's review this example:

```
static UINT GetSize(const std::string& rStr)
{
    return (strlen(rStr.c_str()) + 1);
}
```

This code belongs to a real-life application. Usually such funny code fragments are created during careless refactoring. This code is slow and, even more, quite possibly even unnecessary. When it is required you can just write `"string::length() + 1"`.

Nevertheless, if you are willing to create a special function for calculating the size of a null-terminated string, it should appear as follows:

```
inline size_t GetSize(const std::string& rStr)
{
    return rStr.length() + 1;
}
```

Remark

One should remember that `"strlen(MyString.c_str())"` and `"MyString.length()"` operations will not always generate the same result. The differences will appear in case a string contains null characters besides the terminal one. However such situations can be viewed as a very bad design practice, so the V806 warning message is a great reason to consider the possibility of refactoring. Even if the developer who created this code understands its' operational principles quite well, nevertheless it will be hard to understand this code for his colleagues. They will wonder about the purpose of such a style and could potentially replace the call to

"strlen()" function with "length()", thus creating a bug in the program. So one should not be lazy and should replace it with such a code in which operational principles are clear and intelligible to even an outsider developer. For instance, if the string contains null characters, then there is a high probability that it is not a string at all but an array of bytes. An in such a case the std::vector or your own custom classes should be used instead.

V807. Decreased performance. Consider creating a pointer/reference to avoid using the same expression repeatedly.

The analyzer has detected code which can be optimized. The code contains homogeneous message chains intended to get access to some object. The following constructs are understood by a message chain:

- Get(1)->m_point.x
- X.Foo().y
- next->next->Foo()->Z

If a message chain is repeated more than once, perhaps you should consider code refactoring.

Look at this example:

```
Some->getFoo()->doIt1();
Some->getFoo()->doIt2();
```

If the 'getFoo()' function works slowly or if this code is placed inside a loop, you should rewrite this code. For example, you may create a temporary pointer:

```
Foo* a = Some->getFoo();
a->doIt1();
a->doIt2();
```

Of course, it is not always possible to write it in this way. And moreover, such refactoring does not always give you a performance gain. There exist too many various alternatives, so we cannot give you any general recommendations.

But presence of message chains usually indicates careless code. To [improve such code](#) you can use several methods of refactoring:

1. [Hide Delegate](#)
2. [Extract Method](#)
3. [Move Method](#)

V1000. Did you forget to enable the /openmp compiler option?

The analyzer has detected an error caused by the fact that the "/openmp" compiler option is disabled in your project's properties. When the "/openmp" compiler option is disabled, all OpenMP directives are ignored by the Visual C++ compiler. OpenMP support can be enabled in the "Configuration Properties | C/C++ | Language" section of the project properties dialog.

V1001. Missing 'parallel' keyword.

The analyzer detected a possible error caused by a missing "parallel" keyword in an OpenMP directive. Such incomplete directives are ignored by compiler, and they are not executed as the result. The following sample generates V1001:

```
#pragma omp for
for(int i = 0; i < 4; i++)
{
    #pragma omp critical
    {
        str << "\r\nThread# " << omp_get_thread_num()
        << ": Iteration# " << i;
    }
}
```

Note that this code will be executed only by a single thread, because the "#pragma omp for" directive is incomplete. Here is the correct form of the previous example:

```
#pragma omp parallel for
for(int i = 0; i < 4; i++)
{
    #pragma omp critical
    {
        str << "\r\nThread# " << omp_get_thread_num()
        << ": Iteration# " << i;
    }
}
```

However, if the "#pragma omp for" directive is used in a parallel section, the syntax is correct. Let us consider another sample:

```
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < 4; i++)
    {
        #pragma omp critical
        {
            str << "\r\nThread# " << omp_get_thread_num()
            << ": Iteration# " << i;
        }
    }
}
```

The behavior of this code will be absolutely the same as the behavior of the previous correct sample. Note that both correct samples will not cause V1001, because the directive is complete in both cases.

Here is one more sample which generates V1001:

```
#pragma omp sections
{
    #pragma omp sections
    {
        #pragma omp critical
        {
            str << "\r\nThread# " << omp_get_thread_num();
        }
    }
}
```

```

#pragma omp section
{
    #pragma omp critical
    {
        str << "\r\nThread# " << omp_get_thread_num()
    }
}

```

Since the "#pragma omp sections" directive is incomplete, this code will be executed only by a single thread. Just like in the first sample, the error does not occur if the directive contains the "parallel" keyword or if it is located in a parallel section.

V1002. Missing 'omp' keyword.

The analyzer has detected an error caused by a missing "omp" keyword in an OpenMP directive. The Visual C++ compiler ignores such incomplete directives and they are not executed as the result.

```

#pragma omp parallel num_threads(2)
{
    #pragma single
    {
        str << "A";
    }
}

```

The code above will output the "A" letter to the string stream twice because the "#pragma single" directive is incomplete. The "omp" keyword must be added to the directive in order to fix the problem:

```

#pragma omp parallel num_threads(2)
{
    #pragma omp single
    {
        str << "A";
    }
}

```

Since the "omp" keyword is required in any OpenMP directive, the error can occur in any of the following directives: parallel, sections, for, ordered, section, master, critical, barrier, atomic, flush, etc. If the "omp" keyword is missing, all the directives will be ignored by the Visual C++ compiler.

V1003. Missing 'for' keyword. Each thread will execute the entire loop.

The analyzer has detected a possible error caused by a missing "for" keyword in the "#pragma omp parallel for" OpenMP directive. Let us consider a sample code in which this may lead to an unexpected behavior:

```

#pragma omp parallel num_threads(2)
for(int i = 0; i < 2; i++)
{
    #pragma omp critical
    {
        str << "\r\nThread# " << omp_get_thread_num()
        << ": Iteration# " << i;
    }
}

```

Since the directive lacks the "for" keyword, the work will not be shared between threads and each thread will execute all iterations of the loop. As the result, the loop will be executed 4 times instead of the expected 2 times. Here is the correct version of the code:

```

#pragma omp parallel for num_threads(2)
for(int i = 0; i < 2; i++)
{
    #pragma omp critical
    {
        str << "\r\nThread# " << omp_get_thread_num()
        << ": Iteration# " << i;
    }
}

```

If this behavior is the expected one for your code, and you need each thread to execute all iterations of the loop, you can suppress V1003 by specifying the parallel section explicitly:

```

#pragma omp parallel num_threads(2)
{
    for(int i = 0; i < 2; i++)
    {
        #pragma omp critical
        {
            str << "\r\nThread# " << omp_get_thread_num()
            << ": Iteration# " << i;
        }
    }
}

```

V1004. Nested parallelization of a 'for' loop.

The analyzer has detected a possible error caused by nested parallelism. Let us consider an example in which this may lead to an unexpected behavior:

```

#pragma omp parallel num_threads(2)
{
    // ...
    // N code lines
    // ...
    #pragma omp parallel for
    for(int i = 0; i < 2; i++)
    {
        #pragma omp critical
        {
            str << "\r\nThread# "
            << omp_get_thread_num()
            << ": Iteration# " << i;
        }
    }
}

```

Using the "parallel" keyword in a parallel section leads to nested parallelism. In the sample above each of the two initial threads will be parallelized once again and the loop execution will be shared between two pairs of threads. As the result, if nested parallelism is disabled in your code, the loop will be executed 4 times (once per each thread). You can omit the "parallel" keyword in the "#pragma omp parallel for" in order to avoid this.

```
#pragma omp parallel num_threads(2)
{
    // ...
    // N code lines
    // ...
    #pragma omp for
    for(int i = 0; i < 2; i++)
    {
        #pragma omp critical
        {
            str << "\r\nThread# " << omp_get_thread_num()
                << ": Iteration# " << i;
        }
    }
}
```

If the behavior is the expected one for your code and you need nested parallelism, you can suppress V1004 using the [Detectable Errors](#) settings section.

V1005. The 'ordered' directive is not present in an ordered loop.

The analyzer has detected an error caused by a missing "#pragma omp ordered" directive in a ordered loop. Let us consider an incorrect code:

```
#pragma omp parallel for ordered
for(int i = 0; i < 4; i++)
{
    #pragma omp critical
    {
        str << "\r\nThread# "
            << omp_get_thread_num()
            << ": Iteration# " << i;
    }
}
```

If the "ordered" clause is present in a "#pragma omp parallel for" or "#pragma omp for" directive, the corresponding loop' body must contain the "#pragma omp ordered" directive which specifies the code block in which the loop's iterations will be executed sequentially.

```
#pragma omp parallel for ordered
for(int i = 0; i < 4; i++)
{
    #pragma omp ordered
    {
        #pragma omp critical
        {
            str << "\r\nThread# "
                << omp_get_thread_num()
                << ": Iteration# " << i;
        }
    }
}
```

V1006. Missing omp.h header file. Use '#include <omp.h>'.

The analyzer has detected a strange situation provoked by the absence of header file <omp.h> in the file where OpenMP directives are used.

The absence of the header file in the file where OpenMP directives are used, e.g. such as "#pragma omp parallel for" is not an error but poor style.

Also, if the program uses OpenMP, it should import vcomp.lib /vcompd.lib. Otherwise, an error at the stage of execution will occur. The import of this library is carried out in. That is why, if the import of the needed libraries is not specified in a project explicitly, then #include must be present at least in one file of the project.

V1101. Redefining number of threads in a parallel code.

The analyzer has detected an error caused by an attempt to redefine the number of threads in a parallel section. Let us consider an example of an incorrect code:

```
#pragma omp parallel
{
    omp_set_num_threads(2);
    #pragma omp parallel
    {
    }
}
```

The omp_set_num_threads function changes the number of threads for the next parallel section. However, if the function is called from a parallel section, its behavior is undefined according to the OpenMP 2.0 standard. If you need to specify the number of threads for a nested parallel section, you can use the num_threads clause:

```
#pragma omp parallel
{
    #pragma omp parallel num_threads(2)
    {
    }
}
```

V1102. Non-symmetrical use of set/unset functions for the following lock variable(s): foo.

The analyzer has detected an error caused by a non-symmetrical locks usage. Since locks are similar to critical sections, every thread which uses them must call both locking and unlocking functions. The following code, for example, will cause infinite waiting unless the lexically first section is not executed last:

```
omp_lock_t myLock;
omp_init_lock(&myLock);
#pragma omp parallel sections
{
    #pragma omp section
    {
```

```

        omp_set_lock(&myLock);
    }
#pragma omp section
{
    omp_set_lock(&myLock);
    omp_unset_lock(&myLock);
}
}

```

The rule provided above also means that a thread cannot unlock a variable owned by another thread:

```

omp_lock_t myLock;
omp_init_lock(&myLock);
#pragma omp parallel sections
{
    #pragma omp section
    {
        omp_set_lock(&myLock);
    }
    #pragma omp section
    {
        omp_unset_lock(&myLock);
    }
}

```

This code can cause a run-time error or make the application hang. One should call locking/unlocking functions in a symmetrical way:

```

omp_lock_t myLock;
omp_init_lock(&myLock);
#pragma omp parallel sections
{
    #pragma omp section
    {
        omp_set_lock(&myLock);
        omp_unset_lock(&myLock);
    }
    #pragma omp section
    {
        omp_set_lock(&myLock);
        omp_unset_lock(&myLock);
    }
}

```

V1103. Threads number dependent code. The 'omp_get_num_threads' function is used in an arithmetic expression.

The analyzer has detected a potential error caused by dependency of behavior on the number of threads which execute the code. Using the number of threads in arithmetic operations is unsafe, because the number of threads is equal to the number of virtual CPUs by default in the Visual C++ implementation of OpenMP. Let us consider an incorrect code which is supposed to output all English letters to a string:

```

#pragma omp parallel
{
    int lettersPerThread = 26 / omp_get_num_threads();
    int thisThreadNum = omp_get_thread_num();
    int startLetter = 'a' + thisThreadNum * lettersPerThread;
    int endLetter = 'a' + thisThreadNum * lettersPerThread + lettersPerThread;
    for (int i = startLetter; i < endLetter; i++)
    {
        #pragma omp critical
        {
            str << char(i);
        }
    }
}

```

When the code is executed on a machine with four virtual CPUs, the work will be shared between four threads. Since 26 cannot be divided by 4 without a remainder, only 24 letters out of 26 would be printed in this case. One should exclude the number of threads from arithmetical operations in order to fix the problem (this operation will also simplify the code in this particular case):

```

#pragma omp parallel for
for (int i = 'a'; i <= 'z'; i++)
{
    #pragma omp critical
    {
        str << char(i);
    }
}

```

If you really need to use the number of threads in your calculations, you must make sure that the corresponding number will remain the same regardless of the environment:

```

const int numThreads = 2;
#pragma omp parallel num_threads(numThreads)
{
    int lettersPerThread = 26 / numThreads();
    int thisThreadNum = omp_get_thread_num();
    int startLetter = 'a' + thisThreadNum * lettersPerThread;
    int endLetter = 'a' + thisThreadNum * lettersPerThread + lettersPerThread;
    for (int i = startLetter; i < endLetter; i++)
    {
        #pragma omp critical
        {
            str << char(i);
        }
    }
}

```

If you need to suppress V1103, you can use a temporary variable:

```

int threads = omp_get_num_threads();
int lettersPerThread = 26 / threads;

```

V1104. Redefining nested parallelism in a parallel code.

The analyzer has detected an error caused by an attempt to enable/disable nested parallelism in a parallel section. Let us consider an incorrect code:

```
#pragma omp parallel
{
    omp_set_nested(1);
}
```

According to the OpenMP 2.0 standard, nested parallelism can only be enabled/disabled in a sequential code. If the `omp_set_nested` function is called from a parallel section, the result is undefined. The correct version of the code is provided below:

```
omp_set_nested(1);
#pragma omp parallel
{
}
```

V1201. Concurrent usage of a shared resource via an unprotected call of the 'foo' function.

The analyzer has detected an error caused by concurrent usage of a shared resource in a parallel section. Let us consider an example of incorrect code:

```
#pragma omp parallel num_threads(2)
{
    printf("Hello, world!\r\n");
}
```

Since working with the standard output stream (which is the shared resource in this case) is not an atomic operation, both parallel threads will output their characters simultaneously. As the result, the output will be similar to the following text:

HellHell oo WorWlodrrd

To make the function call safe, you can use critical sections:

```
#pragma omp parallel num_threads(2)
{
    #pragma omp critical
    {
        printf("Hello, world!\r\n");
    }
}
```

One more solution is to use locks. Please note, however, that locks work slower than critical sections. Let us consider one more correct version of the code:

```
omp_lock_t lock;
omp_init_lock(&lock)
#pragma omp parallel num_threads(2)
{
    omp_set_lock(&lock);
    printf("Hello, world!\r\n");
    omp_unset_lock(&lock);
}
```

Since the problem only occurs when the function is called from two or more parallel threads, the following code is correct too:

```
#pragma omp parallel num_threads(2)
{
    #pragma omp single
    {
        printf("Hello, world!\r\n");
    }
}
```

V1202. The 'flush' directive should not be used for the 'foo' variable, because the variable has pointer type.

The analyzer has detected an error caused by using the `"#pragma omp flush"` directive with a pointer. Using the directive with a pointer variable is meaningless, since the operation affects only the stored memory address, not the referenced memory. Let us consider a sample code which reads data from a hardware in one thread and displays the data in another thread:

```
int* a = new int[1];
a[0] = 0;
int flag = 0;
#pragma omp parallel sections
{
    #pragma omp section
    {
        my_read_func(a);
        #pragma omp flush(a)
        flag = 1;
        #pragma omp flush(flag)
    }
    #pragma omp section
    {
        while (flag != 1)
        {
            #pragma omp flush(flag)
        }
        #pragma omp flush(a)
        str << "\r\n" data = " << a[0];
    }
}
delete[] a;
```

Since only the pointer, which stores the address of the first array element, is synchronized, the value in memory may not be synchronized with a read buffer or with a cached value. In order to synchronize the entire shared array the "flush" directive must be applied to the entire memory, not to a single pointer variable:

```
int* a = new int[1];
a[0] = 0;
int flag = 0;
#pragma omp parallel sections
{
    #pragma omp section
    {
```

```

    my_read_func(a);
    #pragma omp flush
    flag = 1;
    #pragma omp flush(flag)
}
#pragma omp section
{
    while (flag != 1)
    {
        #pragma omp flush(flag)
    }
    #pragma omp flush
    str << "\r\ndata = " << a[0];
}
}
delete[] a;

```

V1203. Using the 'threadprivate' directive is dangerous, because it affects the entire file. Use local variables or specify access type for each parallel block explicitly instead.

The analyzer has detected a "#pragma omp threadprivate" directive. The directive is dangerous because it affects the entire file, not a local code block. Let us consider an example of incorrect code:

```

int threadprivate_var;
#pragma omp threadprivate(threadprivate_var)
...
threadprivate_var = 0;
#pragma omp parallel num_threads(4)
{
    #pragma omp atomic
    threadprivate_var++;
    #pragma omp barrier
}

```

Since the threadprivate_var variable is declared as threadprivate, it will be treated as local in the parallel section. As the result, a developer can simply forget about the directive and consider the variable to be shared by default. The easiest workaround which does not change the entire source file's behavior is to use a temporary variable:

```

threadprivate_var = 0;
int b = threadprivate_var;
#pragma omp parallel num_threads(4)
{
    #pragma omp atomic
    b++;
    #pragma omp barrier
}
threadprivate_var = b;

```

This approach, however, will only allow to work around the problem in a single part of the file and does not guarantee that the problem will never occur in other parts of the file later. The safest solution is to reject using the threadprivate directive and create local variables in every thread or, if it is really necessary, make variables declared above local for each specific parallel section by specifying access mode explicitly. If you are really sure that the "#pragma omp threadprivate" directive is required in your code, you can suppress V1203 using the [Detectable Errors](#) settings section.

V1204. Data race risk. Unprotected static variable declaration in a parallel code.

The analyzer has detected an error caused by an unprotected static variable declaration in a parallel section. Since the variable is shared for both threads and both threads work with it concurrently, one of the threads can read its default value before another thread initializes the variable with the expected value. As the result, different threads will display different variable values:

```

#pragma omp parallel num_threads(2)
{
    static int cachedResult = ComputeSomethingSlowly();
    int res = cachedResult;
    ...
}

```

In order to make the declaration safe, the initializing function call and the subsequent variable usage must be placed in a critical section:

```

#pragma omp parallel num_threads(2)
{
    int res;
    #pragma omp critical
    {
        static int cachedResult = ComputeSomethingSlowly();
        res = cachedResult;
    }
    ...
}

```

V1205. Data race risk. Unprotected concurrent operation with the 'foo' variable.

The analyzer has detected an error caused by concurrent operation with a shared variable. Let us consider a sample incorrect code:

```

int a = 0;
#pragma omp parallel for num_threads(4)
for (int i = 0; i < 100000; i++)
{
    a++;
}

```

Since all the threads write to the same memory space and read from the same memory space at the same time, the variable's value after this loop is unpredictable. In order to make the operation safe, one should either place the operation in a critical section or, since the operation in the example above is atomic, use the "#pragma omp atomic" directive. The last option is more preferable since it makes the resulting code faster:

```
int a = 0;
#pragma omp parallel for num_threads(4)
for (int i = 0; i < 100000; i++)
{
    #pragma omp atomic
    a++;
}
```

If the operation's type does not allow using the "#pragma omp atomic" directive, one can use critical sections or locks. Critical sections are more preferable since they work faster:

```
int a = 0;
#pragma omp parallel for num_threads(4)
for (int i = 0; i < 100000; i++)
{
    #pragma omp critical
    {
        ...
    }
}
```

Please note that V1205 will not occur if the variable in question is local or if the code is executed only by a single thread (for example, if the operation is performed in a "#pragma omp single" block).

V1206. Data race risk. The value of the 'foo' variable can be changed concurrently via the 'bar' function.

The analyzer has detected a possible error caused by a concurrent call of a function which takes a shared variable by reference. Since the shared variable is passed by reference, this may lead to a concurrent change of shared data and, therefore, the variable's value may be unpredictable after the function is executed:

```
void dangerousFunction(int &param)
{
    param += 1;
}
...
#pragma omp parallel for
for (int i = 0; i < 100000; i++)
{
    dangerousFunction(a);
}
```

The same error will occur if the function takes a pointer variable and changes the contents of a shared memory space. Since all threads write to the same memory space and read from the same memory space at the same time, the variable's value after the loop is unpredictable.

```
void dangerousFunction2(int *param)
{
    (*param) += 1;
}
...
#pragma omp parallel for
for (int i = 0; i < 100000; i++)
{
    dangerousFunction2(&a);
}
```

In order to make the function's call safe, one should either place it in a critical section or use locks. Critical sections are more preferable since they work faster:

```
#pragma omp parallel for
for (int i = 0; i < 100000; i++)
{
    #pragma omp critical
    {
        dangerousFunction2(&a);
    }
}
```

Please note that V1206 does not occur if the variable is local or if the code is executed only by a single thread (for example, if the code is in the "#pragma omp single" block).

V1207. Data race risk. The 'foo' object can be changed concurrently by a non-const function.

The analyzer has detected a possible error caused by a concurrent call of a non-const method for a shared object. If the method in question changes the object's state, a concurrent call may lead to unpredictable results. Let us consider an example of incorrect code:

```
class MyClass
{
    int a;
public: void nonConstMethod() {a++;};
public: int getA() const {return a;};
};
...
MyClass obj;
#pragma omp parallel for num_threads(2)
for (int i = 0; i < 100000; i++)
{
    obj.nonConstMethod();
}
str << "Result: " << obj.getA() << "\r\n";
```

Since all threads write to the same memory space and read from the same memory space at the same time, the variable's value after the loop is unpredictable. In order to make the function's call safe, one should either place it in a critical section or use locks. Critical sections are more preferable since they work faster:

```
...
#pragma omp parallel for num_threads(2)
for (int i = 0; i < 100000; i++)
{
    #pragma omp critical
    {
        obj.nonConstMethod();
    }
}
```


...

If the method which caused V1207 does not change the object's state, you can suppress V1207 if you declare the method as const:

```
public:
void constMethod() const
{
    ...
}
```

Please note that V1207 does not occur if the variable is local or if the code is executed only by a single thread (for example, if the code is in the "#pragma omp single" block).

V1208. The 'foo' variable of reference type cannot be private.

The analyzer has detected an error caused by declaring a variable of reference type as private. According to the OpenMP 2.0 standard, the operation is invalid:

```
int a = 0;
int& refvar = a;
#pragma omp parallel for num_threads(2) private(refvar)
for (int i = 0; i < 100000; i++)
{
    refvar++;
}
```

If you need to initialize a variable before a parallel section and make the variable local for every thread after this so that every thread inherit the current variable's value, you can use the threadprivate clause:

```
int a = 0;
#pragma omp parallel num_threads(2) firstprivate(a)
{
    for (int i = 0; i < 100000; i++)
    {
        a++;
    }
}
```

V1209. Warning: The 'foo' variable of pointer type should not be private.

The analyzer has detected a possible error caused by declaring a pointer variable as private. The operation is meaningless since every thread will get a copy of an address value, not the referenced memory space, as the result:

```
int* a = new int[1];
a[0] = 0;
#pragma omp parallel firstprivate(a) num_threads(2)
{
    for (int i = 0; i < 100000; i++)
    {
        a[0]++;
    }
    ...
}
delete[] a;
```

When the code is executed, both threads will write to the same shared memory space and read from the same memory space concurrently. As the result, the value of the array element stored in the memory space will be unpredictable. In order to make the entire array private for every thread, the array must be created and initialized in a parallel section. In this case each thread will work with different memory spaces and both threads will get the expected values after the loop execution is over:

```
#pragma omp parallel firstprivate(a) num_threads(2)
{
    int* a = new int[1];
    a[0] = 0;
    for (int i = 0; i < 100000; i++)
    {
        a[0]++;
    }
    ...
}
```

V1210. The 'foo' variable is marked as lastprivate but is not changed in the last section.

The analyzer has detected a possible error caused by the fact that a lastprivate variable is not changed in the last section. According to the OpenMP 2.0 standard, a lastprivate variable takes its resulting value from the lexically last section after the section's execution is over. If the variable is not changed in the last section, this may lead to unpredictable results. The following code, for example, will make an application crash (consider that the lastprivate variable is not changed in the last section):

```
int a;
#pragma omp parallel sections lastprivate(a)
{
    #pragma omp section
    {
        a = 0;
        for (int i = 0; i < 100000; i++)
        {
            a++;
        }
    }
    #pragma omp section
    {
        ...
    }
}
```

The code will work as expected if the sections are swapped:

```
int a;
#pragma omp parallel sections lastprivate(a)
{
    #pragma omp section
    {
```

```

        ...
    }
    #pragma omp section
    {
        a = 0;
        for (int i = 0; i < 100000; i++)
        {
            a++;
        }
    }
}

```

V1211. The use of 'flush' directive has no sense for private 'foo' variable, and can reduce performance.

The analyzer detected a potentially inefficient code containing redundant directives flush. Flush directive has no sense for local variables and variables marked private, threadprivate, lastprivate or firstprivate. The analyzer warns about applying flush directive to such variables.

Redundant use of flush directive does not cause errors but reduces performance. In the demo-program ParallelSample included into PVS-Studio, there is an example showing probable slowdown of the program's operation. The code with redundant flush's operates 20 times slower than without these (see code of functions V1211 and V1211_correct).

Here is an example leading to diagnostic warning V1211:

```

#pragma omp parallel for num_threads(4) firstprivate(a)
for (int i = 0; i < n; i++)
{
    #pragma omp flush(i) // V1211
    #pragma omp flush(a) // V1211
    Array[i] = a;
}

```

To correct the code you must remove all the unnecessary flush directives:

```

#pragma omp parallel for num_threads(4) firstprivate(a)
for (int i = 0; i < n; i++)
{
    Array[i] = a;
}

```

The code where flush directive does have sense is considered correct and no diagnostic warning is shown. For example:

```

int a;
#pragma omp parallel for num_threads(4)
for (int i = 0; i < n; i++)
{
    #pragma omp flush(a)
}

```

V1212. Data race risk. When accessing the array 'foo' in a parallel loop, different indexes are used for writing and reading.

The analyzer detected a potential error in the code related to accessing array items in a parallel loop. The access is performed using different indexes and it may cause a race condition. Check if your code contains an error similar to this one:

```

#pragma omp parallel for
for (ptrdiff_t i = 1; i < len; i++)
{
    arr[i - 1] += arr[i];;
}

```

In this code we add each array item to the following one. The code was parallelized incorrectly, so having the input sequence "1 1 1 1 1" we expect "2 2 2 2 1" but may get "2 2 3 2 2 1" instead.

To correct such errors you should change the algorithm and/or employ synchronization mechanisms.

Here is an example the analyzer considers correct because it does not contain calls to one array with different indexes:

```

int a;
#pragma omp parallel for num_threads(4)
for (int i = 0; i < n; i++)
{
    #pragma omp flush(a)
}

```

V1301. The 'throw' keyword cannot be used outside of a try..catch block in a parallel section.

The analyzer found an error relating to throwing an exception from a parallel block. According to OpenMP specification, if you use exceptions inside a parallel block all these exceptions should be processed inside this block. The example given below leads to incorrect program's behavior and most likely to a program crash:

```

void foo1301(const char **strings, ptrdiff_t n)
{
    #pragma omp parallel for
    for (ptrdiff_t i = 0; i < n; i++)
    {
        if (!strings[i])
            throw MyException();
        DoSomething(strings[i]);
    }
}

```

Correction of the code lies in processing exceptions inside the parallel block and transferring the information about the error through other mechanisms. Below are given two variants of the corrected function:

```

void foo1301_fixed1(const char **strings, ptrdiff_t n)
{

```

```

ptrdiff_t errCount = 0;
#pragma omp parallel for reduction(+: errCount)
for (ptrdiff_t i = 0; i < n; i++)
{
    try
    {
        if (!strings[i])
            throw MyException();
        DoSomething(strings[i]);
    }
    catch (MyException &)
    {
        #pragma omp atomic
        ++errCount;
    }
}
if (errCount != 0)
    throw MyException();
}

void fool301_fixed2(const char **strings, ptrdiff_t n)
{
    size_t errCount = 0;
#pragma omp parallel for reduction(+: errCount)
    for (ptrdiff_t i = 0; i < n; i++)
    {
        if (!strings[i])
            ++errCount;
        else
            DoSomething(strings[i]);
    }
    if (errCount != 0)
        throw MyException();
}

```

V1302. The 'new' operator cannot be used outside of a try..catch block in a parallel section.

The analyzer found an error relating to throwing an exception from a parallel block. According to OpenMP specification, if you use exceptions inside a parallel block all these exceptions should be processed inside this block. If you use "new" operator inside parallel code you should provide interception of the exception which will be generated when an error of memory allocation occurs according to C++ standard. The example given below leads to incorrect program's behavior and most likely to a program crash if an error of memory allocation occurs:

```

void fool302(ptrdiff_t n)
{
    #pragma omp parallel for
    for (ptrdiff_t i = 0; i < n; i++)
    {
        float *array = new float[10000];
        //...
        delete [] array;
    }
}

```

Correction of the code lies in processing exceptions inside the parallel block and transferring the information about the error through other mechanisms or in refusing to use "new" operator. This is the corrected variant of the function:

```

void fool302_fixed(ptrdiff_t n)
{
    #pragma omp parallel for
    for (ptrdiff_t i = 0; i < n; i++)
    {
        try {
            float *array = new float[10000];
            //...
            delete [] array;
        }
        catch (std::bad_alloc &) {
            // process exception
        }
    }
}

```

V1303. The 'foo' function which throws an exception cannot be used in a parallel section outside of a try..catch block.

The analyzer found an error relating to throwing an exception from a parallel block. According to OpenMP specification, if you use exceptions inside a parallel block all these exceptions should be processed inside this block. The analyzer warns about the call of a function which is marked as throwing exceptions in a parallel block and which is not protected by try..catch block.

When an exception is thrown from ExceptionFoo function the example given below leads to incorrect program's behavior and most likely to a program crash:

```

void ExceptionFoo() throw(...) { }
void fool303(ptrdiff_t n)
{
    #pragma omp parallel for
    for (ptrdiff_t i = 0; i < n; ++i)
    {
        //...
        ExceptionFoo();
        //...
    }
}

```

Correction of the code lies in processing exceptions inside the parallel block and transferring information about the error through other mechanisms. Below the two variants of the corrected function are given:

```

void fool303_fixed(ptrdiff_t n)
{
    #pragma omp parallel for
    for (ptrdiff_t i = 0; i != n; ++i)
    {
        try {
            //...

```

```

        ExceptionFoo();
        //...
    }
    catch (...) {
        // process exception
    }
}

```

You should keep in mind that functions not marked as throw(...) can also generate exceptions. But VivaMP analyzer doesn't consider calling them unsafe. It is made to generate diagnostic messages in a reasonable number. Otherwise any code containing function call will be considered unsafe. The following principle is used:

void foo(); - suppose it don't throw an exception

void foo() throw(); - doesn't throw an exception

void foo() throw(...); - throws an exception

V2001. Consider using the extended version of the 'foo' function here.

This diagnostic warning was added on users' request.

The analyzer allows you to detect calls of functions that have "extended" analogues. By the term "extended functions" we understand functions that have the Ex suffix. Here are some examples of extended functions: VirtualAllocEx, SleepEx, GetDCEx, LoadLibraryEx, FindResourceEx.

Consider the following source code:

```

void foo();
void fooEx(float x);
void foo2();
...
void test()
{
    foo(); // V2001
    foo2(); // OK
}

```

In the fragment where the "foo" function is called, the V2001 diagnostic message will be produced since there is another function with the same name but ending with "Ex". The "foo2" function does not have an alternative version and therefore no diagnostic message will be generated concerning it.

The V2001 message will be also generated in the following case:

```

void fooA(char *p);
void fooExA(char *p, int x);
...
void test()
{
    fooA(str); // V2001
}

```

[V2002](#) is a related diagnostic message.

V2002. Consider using the 'Ptr' version of the 'foo' function here.

This diagnostic message was added on users' request.

The analyzer allows you to detect calls of functions that have 'Ptr' analogues. By this term we mean functions whose name has the Ptr suffix. Here are some examples of extended functions: SetClassLongPtr, DSA_GetItemPtr.

Consider the following source code:

```

void foo(int a);
void fooPtr(int a, bool b);
void foo2();
...
void test()
{
    foo(1); // V2002
    foo2(); // OK
}

```

In the fragment where the "foo" function is called, the V2002 diagnostic message will be produced since there is another function with the same name but ending with "Ptr". The "foo2" function does not have an alternative version and therefore no diagnostic message will be generated concerning it.

The V2002 message will be also generated in the following case:

```

void fooA(char *p);
void fooPtrA(char *p, int x);
...
void test()
{
    fooA(str); // V2002
}

```

[V2001](#) is a related diagnostic message.

V2003. Explicit conversion from 'float/double' type to signed integer type.

This diagnostic warning was added at the request of users.

The analyzer allows you to detect all the explicit floating-point type conversions to integer signed types.

Consider some examples of constructs the analyzer will generate this diagnostic message on:

```

float f;
double d;
long double ld;
int i;

```

```

short s;
...
i = int(f); // V2003
s = static_cast<short>(d); // V2003
i = (int)ld; // V2003

```

V2004 is a related diagnostic message.

V2004. Explicit conversion from 'float/double' type to unsigned integer type.

This diagnostic warning was added at the request of users.

The analyzer allows you to detect all the explicit floating-point type conversions to integer unsigned types.

Consider some examples of constructs the analyzer will generate this diagnostic message on:

```

float f;
double d;
long double ld;
unsigned u;
size_t s;
...
u = unsigned(f); // V2004
s = static_cast<size_t>(d); // V2004
u = (unsigned)ld; // V2004

```

V2003 is a related diagnostic message.

V2005. C-style explicit type casting is utilized. Consider using: static_cast/const_cast/reinterpret_cast.

This diagnostic warning has been added at the request of users.

The analyzer allows you to detect explicit type conversions written in the old C language style in a C++ program. It is safer in the C++ language to convert types using operators static_cast, const_cast and reinterpret_cast.

The V2005 diagnostic rule helps to perform code refactoring and replace the old type conversion style with a new one. Sometimes it helps to detect errors.

Here are examples of constructs that will trigger this diagnostic message:

```

int i;
double d;
size_t s;
void *p;
...
i = int(p); //V2005
d = (double)d; //V2005
s = (size_t) i; //V2005

```

The V2005 diagnostic message is not generated in three cases.

1. This is a C program.
2. The conversion target type is void. This type conversion is safe and is used to emphasize that there is a result which is not used anyhow. For example:

```
(void) fclose(f);
```

3. The type conversion is located inside a macro. If the analyzer generated the warning for macros, there would be a lot of reports when different system constants and macros are used. And you cannot fix them anyway. Here you are some examples:

```

#define FAILED(hr) ((HRESULT)(hr) < 0)
#define SRCCOPY (DWORD)0x00CC0020
#define RGB(r,g,b)\
((COLORREF) (((BYTE)(r) | ((WORD)((BYTE)(g))<<8))\
| (((DWORD)(BYTE)(b))<<16)))

```

References

1. Terminology. Explicit type conversion. <http://www.viva64.com/en/t/0015/>
2. Wikipedia. Type conversion. <http://www.viva64.com/go.php?url=742>

Credits and acknowledgements

Windows, Visual Studio, Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Other product and company names mentioned herein may be the trademarks of their respective owners.

PVS-Studio uses SourceGrid control (sourcegrid.codeplex.com). Below you can read Source Grid License.

SourceGrid LICENSE (MIT style)

SourceGrid LICENSE (MIT style)

Copyright (c) 2009 Davide Icardi

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Portions of PVS-Studio are based in part of OpenC++. Bellow you can read OpenC++ Copyright Notice.

*** Copyright Notice

Copyright (c) 1995, 1996 Xerox Corporation.

All Rights Reserved.

Use and copying of this software and preparation of derivative works based upon this software are permitted. Any copy of this software or of any derivative work must include the above copyright notice of Xerox Corporation, this paragraph and the one after it. Any distribution of this software or derivative works must comply with all applicable United States export control laws.

This software is made available AS IS, and XEROX CORPORATION DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, AND NOTWITHSTANDING ANY OTHER PROVISION CONTAINED HEREIN, ANY LIABILITY FOR DAMAGES RESULTING FROM THE SOFTWARE OR ITS USE IS EXPRESSLY DISCLAIMED, WHETHER ARISING IN CONTRACT, TORT (INCLUDING NEGLIGENCE) OR STRICT LIABILITY, EVEN IF XEROX CORPORATION IS ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

*** Copyright Notice

Copyright (C) 1997-2001 Shigeru Chiba, Tokyo Institute of Technology.

Permission to use, copy, distribute and modify this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

Shigeru Chiba makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

*** Copyright Notice

Permission to use, copy, distribute and modify this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Other Contributors make no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

2001-2003 (C) Copyright by Other Contributors.

PVS-Studio can use Clang as preprocessor. Read Clang/LLVM license:

=====

LLVM Release License

=====

University of Illinois/NCSA

Open Source License

Copyright (c) 2007-2011 University of Illinois at Urbana-Champaign.

All rights reserved.

Developed by:

LLVM Team

University of Illinois at Urbana-Champaign

<http://llvm.org>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal with the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimers.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.

* Neither the names of the LLVM Team, University of Illinois at Urbana-Champaign, nor the names of its contributors may be used to endorse or promote products derived from this Software without specific prior written permission.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS WITH THE SOFTWARE.

=====

The LLVM software contains code written by third parties. Such software will have its own individual LICENSE.TXT file in the directory in which it appears. This file will describe the copyrights, license, and restrictions which apply to that code.

The disclaimer of warranty in the University of Illinois Open Source License applies to all code in the LLVM Distribution, and nothing in any of the other licenses gives permission to use the names of the LLVM Team or the University of Illinois to endorse or promote products derived from this Software.

The following pieces of software have additional or alternate copyrights, licenses, and/or restrictions:

Program Directory

<none yet>